UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

GRAMÁTICA TRANSFORMACIONAL COM ATRIBUTOS

por

AVELINO FRANCISCO ZORZO

Dissertação submetida como requisito parcial para
a obtenção do grau de Mestre em
Ciência da Computação

Prof. PAULO ALBERTO DE AZEREDO
Orientador

Porto Alegre, 07 de abril de 1994

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Zorzo, Avelino Francisco

Gramática transformacional com atributos/ Avelino Francisco Zorzo. Porto Alegre: CPGCC da UFRGS, 1994.

100p.:il

Dissertação(mestrado) - Universidade Federal do Rio Grande do Sul, Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, 1994. Orientador: Azeredo, Paulo Alberto.

Dissertação: Gramática Transformacional, Gramática de Atributos, Gramática Transformacional com Atributos, Compilador de Compiladores, Linguagens.

"O mundo pertence aos otimistas;
os pessimistas são meros espectadores."

à minha eterna
namorada MÁRI

AGRADECIMENTOS

Agradeço,

- ao Professor Doutor Paulo Azeredo pela orientação e pela paciência que dispendeu durante este trabalho;

- ao Felipe Schaan de Quadros que me incentivou no Trabalho Individual, do qual surgiram idéias para este trabalho;

- à Mári, minha esposa, que sempre me incentivou a terminar este trabalho;

- a todos aqueles que de uma forma ou outra contribuiram para que este trabalho fosse completado da melhor maneira possível.

RESUMO

A transformação entre linguagens, ou entre diferentes formatos de uma mesma linguagem, é um assunto que desperta interesse há vários anos e desta forma alguns trabalhos tem surgido para tentar automatizar o processo de transformação entre notações diferentes.

Este trabalho descreve as Gramáticas Transformacionais empregados para descrever as transformações necessárias para converter uma notação em uma linguagem fonte (LF) para uma notação equivalente em uma linguagem objeto (LO). Nesta Gramática é embutido o conceito de Gramáticas de Atributos, criando assim as Gramáticas Transformacionais com Atributos (GTAs).

Para validação das GTAs é apresentado um protótipo de ferramenta transformacional, que gera um tradutor, de LF para LO, a partir da descrição da gramática da LF e das regras de transformações para a LO. Tanto a LF quanto a LO são gramáticas do tipo LALR(1).

Com o objetivo de construir a ferramenta mais genérica possível, foram realizados estudos sobre três ferramentas, com as quais as transformações são possíveis. São elas: YACC, SINLEX e GG. É feita uma breve descrição destas três ferramentas e uma comparação com o protótipo implementado.

PALAVRAS-CHAVE: Gramática Transformacional, Gramática de Atributos, Gramática Transformacional com Atributos, Compiladores.

TITLE: "ATTRIBUTED TRANSFORMATIONAL GRAMMAR"

ABSTRACT

Languages transformation or transformation among differents formats of the same language is a subject that has had a lot of interest for many years. Thus, research has been done aiming to automatize the proccess of transformation from one notation to another.

This work describes the use of Transformation Grammars to describe the necessary transformations to convert from a Source Language (SL) notation to an equivalent Object Language (OL). The concept of Attribute Grammars is embbeded to these grammars, defining an Attributed Transformation Grammar (ATG).

A transformation tool prototype to evaluate the ATGs is presented. This tool generates a translator from SL to OL using the SL grammar description and the corresponding transformation rules to the OL. Both the SL and OL are LALR(1) grammars.

Studies on YACC, SINLEX and GG (tools wich allow transformations) were done trying to reach the most generic tool. A brief descriptions of these tools and a comparision with the prototype is presented.

KEYWORDS: Transformational Grammar, Attributed Grammar, Attributed Transformational Grammar, Compilers

SUMÁRIO

LISTA DE ABREVIATURAS

GT - Gramática Transformacional

GA - Gramática de Atributos

GTA - Gramática Transformacional com Atributos

BNF - Backus Naur Form

GLC - Gramática Livre do Contexto

LDGTA - Linguagem de Descrição de Gramáticas Transformacionais com Atributos

UCC - Um Compilador de Compiladores

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE GRAMÁTICAS

1. INTRODUÇÃO

Desde o surgimento dos computadores eletrônicos, muitos pesquisadores tem concentrado esforços na tradução de linguagens naturais e de linguagens formais. O sucesso destas pesquisas se deve, e muito, ao grande uso dos computadores atualmente. Em vez de os usuários programarem em linguagem de máquina, uma tarefa cansativa e com uma taxa de erros muito grande, usam-se linguagens abstratas de alto nível, que resolvem seus problemas. Programas escritos nestas linguagem podem ser automaticamente convertidos para instruções em linguagem de máquina, graças a algumas técnicas de tradução.

Tradutores estão sendo cada vez mais utilizados em muitas áreas. Em vez de serem utilizados somente para descrever regras para compiladores e montadores, é comum utilizar tradutores como processador de comandos, como ferramenta para para desenhar figuras, e como "Compiladores de Silício", programas que produzem leiautes de baixo nível de circuitos eletrônicos a partir de descrições abstratas. As técnicas de tradução são aplicadas a uma grande quantia de problemas. [YEL88]

Nos primeiros dias da Ciência da Computação, as pesquisas sobre tradução centravam-se na descrição de *regras de sintaxe*. Sobre este assunto existe uma grande variedade de estudos (ver, por exemplo, [AHO72]). Depois de desenvolver um formalismo para descrever a estrutura de sentenças da linguagem, pesquisadores desenvolveram algoritmos para determinar se uma dada sentença é ou não válida para uma dada linguagem. Através destes algoritmos construíram-se programas chamados *parsers*. Estes programas tomam uma descrição declarativa da sintaxe de uma linguagem e automaticamente produzem um eficiente programa que reconhece todas as sentenças válidas para aquela linguagem.

Mais recentemente, os pesquisadores tem concentrado esforços na semântica das linguagens. Depois de descobrir muitos caminhos de descrever o significado e traduções de liguagens de uma maneira precisa, pesquisadores iniciaram a analisar características destes formalismos. Estas teorias novamente auxiliaram na criação de ferramentas para o processo de tradução. O resultado foi a criação dos chamados *compiler compilers*. Estes programas tomam uma descrição declarativa de uma tradução e produzem um programa que realiza a tradução de acordo com a especificação. Desde que a construção de um tradutor de maneira manual pode trazer um grande esforço, estes programas converteram-se em uma técnica de tradução muito utilizada. Não somente por automatizar o processo de desenvolvimento de software, mas também por garantir que o tradutor faz exatamente a tarefa para a qual ele foi especificado.

Este trabalho apresenta uma forma de descrever traduções entre linguagens e introduz uma nova ferramenta para o processo de tradução. Uma forma de descrever estas traduções é a utilização de Gramáticas Transformacionais, de modo a especificar tradutores uni-direcionais. Porém, Gramáticas Transformacionais não conseguem resolver todos os problemas de tradução que surgem, sendo necessário anexar às suas características um formalismo com poder computacional maior. Uma forma de aumentar a utilização é anexar atributos ao conceito de Gramáticas Transformacionais, criando assim as Gramáticas Transformacionais com Atributos. Desta forma produz-se uma gramática com poder computacional equivalente às Máquinas de Turing, uma vez que Gramáticas de Atributos possuem poder computacional equivalente as Máquinas de Turing [YEL88]. Como uma GTA pode ser escrita somente como uma GA então esta afirmação é válida.

No Capítulo 2 deste trabalho são apresentados fundamentos de Gramáticas Transformacionais com suas características e formas de transformações. No Capítulo 3 são apresentadas as Gramáticas de Atributos com suas características e definições. No Capítulo 4 é apresentada a utilização das Gramáticas Transformacionais com Atributos para o processo de transformações entre linguagens. Três ferramentas são descritas no Capítulo 5, com enfoque para sua utilização como ferramentas transformacionais. No Capítulo 6 é apresentado o protótipo de uma ferramenta transformacional que utiliza as GTAs. No Capítulo 7 é feita a validação da ferramenta. Finalmente no Capítulo 8 são sugeridos dois novos estudos envolvendo GTAs.

2. GRAMÁTICAS TRANSFORMACIONAIS

Desde a publicação de "Syntactic Structures" de Noam Chomsky, em 1957 [CHO57], tem aumentado o estudo do ramo da teoria linguística que é conhecido como "Gramática Transformacional". Este interesse pode ser visualizado pela quantia de trabalhos publicados sobre este assunto, [BAC64], [KOU66], [DER74], [CAS90], [QUA91], [ZOR91]. Além dos trabalhos publicados, a teoria a ser desenvolvida sobre o assunto tamém é intensa.

Uma gramática transformacional é uma gramática que descreve transformações dentro de uma mesma linguagem, ou entre linguagens diferentes, tendo sido inicialmente utilizada na área da linguística para descrever equivalências de uma mesma língua [SAL73].

Como exemplo, pode-se escrever com uma GT as tranformações necessárias para passar uma frase em português da voz ativa para a voz passiva, e vice-versa. Porém, pode-se generalizar uma GT para transformações entre linguagens diferentes. Por exemplo, na área de Compiladores pode-se usar GTs para converter uma linguagem fonte para uma linguagem objeto; ou ainda, na área de Linguagem Natural, para a transformação da Língua Inglesa para a Língua Portuguesa, sendo algumas formas mais complexas que outras.

Uma GT [SAL73] é composta por regras operacionais diferentes de produções normais de gramáticas e que são chamadas regras de transformação. Estas regras operam em sentenças estruturadas gerando sentenças estruturadas, ou seja, operam em árvores para produzir outras árvores.

Uma regra de transformação possui duas partes: a sentença de domínio e a de mudança estrutural. A sentença de domínio é uma produção que define como a árvore será produzida e a condição para que a tranformação seja efetuada. A sentença de mudança estrutural define as operações a serem executadas na árvore, ou seja, descreve como obter a árvore transformada.

Existem diversas maneiras de representar uma regra de transformação. Dentre as existentes destacam-se as apresentadas em [KOU66].

Forma 1:

```
S ::= A B C
Transformação: x1 x2 x3  -> x3 x2 x1
```

Onde "x" indica um dos símbolos do lado direito da produção, o índice indica a posição do símbolo na produção, e o símbolo "->" indica a transformação desejada.

Forma 2:

```
S ::= A B C  ->  3 2 1
```

Onde os números representam as posições dos símbolos do lado direito da produção, isto é, o indice 1 representa o símbolo A, 2 o símbolo B e 3 o símbolo C. O símbolo "->" indica a transformação desejada. Este formato é adotado em [QUA91].

Forma 3:

```
S ::= A B C  -> C B A
```

Sendo esta que será adotada neste trabalho.

2.1 Transformações possíveis

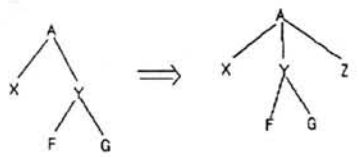
Existem vários tipos de transformações que podem ser realizadas a partir de uma árvore de derivação. A seguir são definidas e exemplificadas as possíveis transformações existentes. Podem, ainda, existir outras transformações a partir da combinação das citadas abaixo.

Quando um símbolo é substituído, removido ou trocado de local no momento da transformação, toda a subárvore de derivação é considerada como um todo.

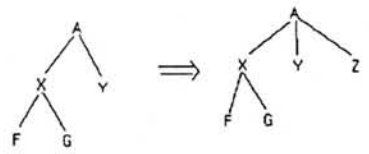
2.1.1 Adição

Um ou mais símbolos são adicionadas em uma árvore de derivação. Os símbolos que são adicionados, são derivados a partir do símbolo raiz da produção. Por exemplo, dada a produçao A ::= X Y -> X Y Z, onde Z é o símbolo adicionado, obtem-se:

se X ::=ε e Y ::= F G

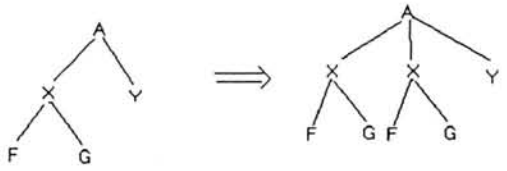


ou, se X ::= F G e Y ::= ε

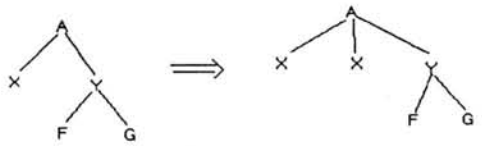


Dada outra regra: A ::= X Y -> X X Y, tem-se:

se X ::= F G e Y ::= ε

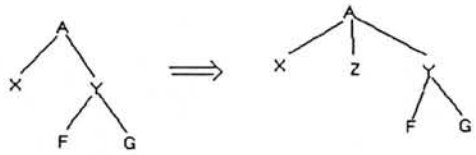


ou, se X ::= ε e Y ::= F G

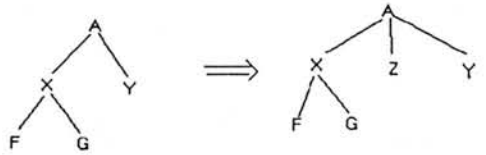


E dada a regra: A ::= X Y -> X Z Y, onde Z é o símbolo a ser inserido, obtem-se:

se X ::= ε e Y ::= F G



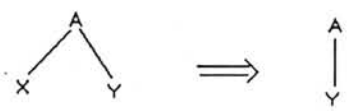
ou, se X ::= F G e Y ::= ε



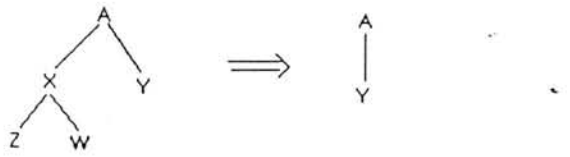
2.1.2 Remoção

Um ou mais símbolos são retirados de uma dada árvore de derivação. Por exemplo, dada a produção: A ::= X Y -> Y, obtem-se,

se X ::= ε



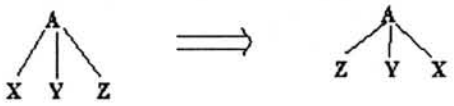
ou, se X ::= Z W



2.1.3 Permutação

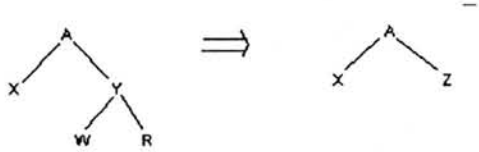
Dois símbolos que pertencem a uma mesma produção são rearranjados dentro da árvore de derivação. No exemplo abaixo, para a produção A ::= X Y -> Y X, os símbolos X e Y são trocados de lugar quando a transformação ocorre.

ou, na produção A :: X Y Z -> Z Y X, os símbolos X e Z são trocados, enquanto o símbolo Y não muda de posição.



2.1.4 Substituição

Um ou mais símbolos de uma árvore de derivação são substituidos ou trocados por um ou mais símbolos. Esta substituição ocorre sempre que um símbolo substitui outro na mesma posição que tal se encontrava. No exemplo, para a produção A ::= X Y -> X Z, o símbolo Y é substituido pelo símbolo Z.



2.2 Esquema de Tradução Dirigida pela Sintaxe

Existem muitas maneiras para especificar um esquema de tradução. Análogo a geradores de linguagens, assim como gramáticas, pode-se especificar sistemas que geram os pares de tradução. Pode-se também reconhecer duas sentenças de entrada com estes dois pares na tradução. Ou pode-se definir um autômato que toma uma sentença de entrada "x" e emite todos "y" tal que "y" é uma tradução de "x".

Em [AHO73] é apresentado um formalismo para definir tradução. Este formalismo é uma gramática na qual os elementos de tradução são associados a cada produção. Deste modo, a produção é usada na derivação da sentença de entrada, enquanto o elemento de tradução é usado para calcular uma porção da sentença de saída associada com a porção da sentença de entrada gerada por aquela produção. Este formalismo foi denominado *Esquema de Tradução Dirigido pela Sintaxe* (ETDS).

Exemplo 1.

Considere o seguinte esquema de tradução que define a tradução (x,x^r) $x \in (0,1)^*$. Isto é, para cada entrada x, a saída é x invertida. As regras que definem esta tradução são:

Produção	Tradução
(1) S -> 0S	S = S0
(2) S -> 1S	S = S1
(3) S -> ε	S = ε

Gramática 2.1 Inversão de número binário

Um par entrada-saída na tradução definida por este esquema pode ser obtido gerando uma sequência de pares de sentenças (α,β) chamado "formas de tradução", onde α é uma sentença de entrada e β uma sentença de saída. Inicia-se com a tradução (S,S). Pode-se aplicar a primeira regra por esta forma. Assim, expande-se o primeiro S usando S->0S. Troca-se então a sentença de saída S por S0 de acordo com o elemento de tradução S=S0. Desta forma pode-se pensar no elemento de tradução como uma produção S->S0. Assim obtem-se a forma de tradução (S0,0S). Pode-se expandir cada S nesta nova forma de tradução usando a regra (1), obtendo assim (00S,S00). Se for aplicada a regra (2), obtem-se (001S,S100). Se for aplicada a regra (3), obtem-se (001,100). Após a aplicação da regra (3) não é possível aplicar mais nenhuma regra, então tem-se (001,100) como um esquema de tradução definida pelas regras acima.

Exemplo 2.

Considere o seguinte esquema de tradução que converte uma expressão aritmética infixada para a notação Polonesa:

	Produção	Tradução
(1)	E->E+T	E=ET+
(2)	E->T	E=T
(3)	T->T*F	T=TF*
(4)	T->F	T=F
(5)	F->(E)	F=E
(6)	F->a	F=a

Gramática 2.2 Conversão de infixada para polonesa

Para determinar a saída para a entrada "a+a*a", primeiro encontra-se a derivação para a sentença de

entrada, E. Calcula-se então a sequência correspondente das traduções como mostrada abaixo:

```
(E,E) ==> (E+T,ET+)
          (T+T,TT+)
          (F+T,FT+)
          (a+T,aT+)
          (a+T*F,aTF*+)
          (a+F*F,aFF*+)
          (a+a*F,aaF*+)
          (a+a*a,aaa*+)
```

Cada forma sentencial da saída é computada pela troca do não-terminal na forma sentencial da saída anterior pelo lado direito da regra de tradução associada com a produçáò usada na derivação da forma sentencial da entrada.

Os esquemas de tradução apresentados nos Exemplos 1 e 2 são casos especiais do *Esquema de Tradução Dirigido pela Sintaxe*.

2.3 Definição de Gramática Transformacional

Uma Gramática Transformacional pode ser considerada um Esquema de Tradução Dirigida pela Sintaxe, pois para cada produção em uma GT existe uma regra transformacional correspondente, ou seja dependendo da sintaxe da sentença de entrada uma determinada transformação será efetuada.

Assim, uma GT é uma 5-upla $T=(N,\Sigma,\Delta,R,S)$, onde:

(1) N é um conjunto finito de símbolos não-terminais;
(2) Σ é um alfabeto finito de entrada;
(3) Δ é um alfabeto finito de saída;
(4) R é um conjunto finito de regras da forma
    $A\rightarrow\alpha,\beta$, onde $\alpha \in (N \cup \Sigma)^*$, $\beta \in (N \cup \Delta)^*$.
(5) S é um não-terminal de N

Em [AHO73], no ETDS, R é definido como um conjunto finito de regras da forma A->α,β, onde $\alpha \in (N \cup \Sigma)^*$, $\beta \in (N \cup \Delta)^*$, e os Não-terminais em β são uma permutação dos Não-terminais de α. O que não é válido para as GTs, pois em β pode haver a inserção de um novo símbolo que não se encontra em α.

3. GRAMÁTICAS DE ATRIBUTOS

Uma Gramática de Atributos (GA) é uma Gramática Livre do Contexto (GLC) [AHO86]. Ela define o domínio livre do contexto sobre o qual está definida a tradução. Esta gramática é aumentada com atributos e funções semânticas. Os atributos são associados aos símbolos da gramática. As funções semânticas são associadas às produções e descrevem como os valores de atributos dos símbolos da produção são obtidos em função dos valores dos outros atributos da produção. Funções semânticas não possuem efeitos colaterais e seus argumentos são constantes ou outras ocorrências de atributos [AHO86].

Sendo N um não-terminal da gramática livre do contexto G que serve como base para a gramática de atributos, denota-se por A(N) o conjunto de atributos associados ao símbolo gramatical N. Neste conjunto existem dois tipos de atributos: sintetizados (S(N)) e herdados (I(N)). Atributos sintetizados propagam informações para cima, em direção à raiz da árvore semântica. Os valores dos atributos sintetizados de N são definidos em produções cujo lado direito é N (ou sejam produções da forma N -> α). Atributos herdados propagam informação para baixo na árvore, em direção às folhas. Os valores dos atributos herdados de N são definidos em produções nas quais N aparece no corpo da produção (ou seja, da forma X -> α N β).

Exemplo:

A definição dirigida pela sintaxe abaixo é de uma calculadora. Associado aos não-terminais E, F e T existe um atributo chamado *val*. Para cada uma das produções E, F

e T, a regra semântica calcula o valor do atributo *val* para o não-terminal do lado esquerdo a partir dos valores dos atributos *val* dos não-terminais do lado direito da produção.

	Produção	Ação Semântica
(1)	L -> E	printf(E.val)
(2)	E -> E_1 + T	E.val = E_1.val + T.val
(3)	E -> T	E.val = T.val
(4)	T -> T_1 * F	T.val = T_1.val * F.val
(5)	T -> F	T.val = F.val
(6)	F -> (E)	F.val = E.val
(7)	F -> id	F.val = id.val

Gramática 3.1 Avaliação de expressão infixada[AHO86]

Dada a expressão 5*3+4, e executadas as ações semânticas, o valor impresso pela ação de (1) é 19. A figura abaixo contém a árvore de derivação com os atributos para a entrada 5*3+4. A saída impressa, quando for executada a ação semântica associada a raiz da árvore, é o valor de E.*val* do primeiro filho da raiz.

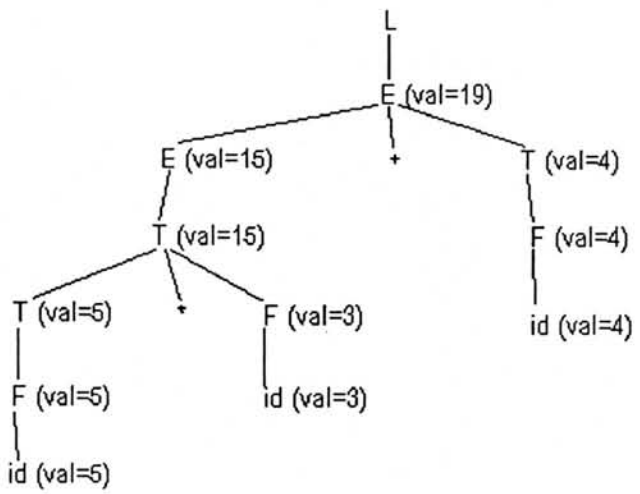


Figura 3.1 Árvore de derivação para 5*3+4

Exemplo:

A gramática abaixo define uma linguagem onde existe um tipo (**int** ou **real**) associado a uma lista de identificadores. O símbolo inicial da gramática abaixo é D. O não-terminal T possui um atributo sintetizado *type*, cujo valor depende do tipo analisado. A regra semântica *L.in=T.type*, associada com a produção **D->T L**, atribui ao atributo herdado *L.in* o valor do tipo. O valor é então passado para baixo na árvore usando o atributo herdado *L.in*. Regras associadas à produção L chamam a rotina *addtype* que adiciona a variável reconhecida em uma tabela de símbolos.

	Produção	Ação semântica
(1)	D -> T L	L.in = T.type
(2)	T -> int	T.type = INTEGER
(3)	T -> real	T.type = REAL
(4)	L -> L1 , id	L1.in = L.in addtype(id.val, L.in)
(5)	L -> id	addtype(id.val, L.in)

Gramática 3.2 Lista de identificadores com tipo [AHO86]

A figura abaixo mostra a árvore de derivação para a sentença "real id, id, id". O valor de *L.in* nos três nodos da árvore é o tipo de cada um dos identificadores "id". Estes valores são calculados a partir do valor do atributo de *T.type*.

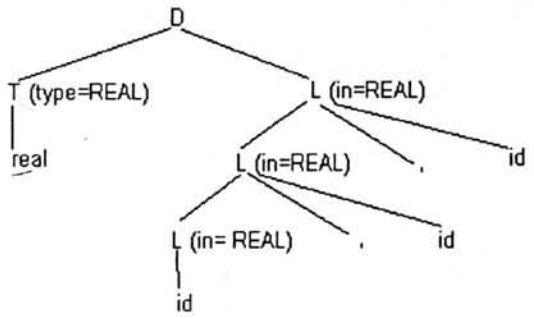


Figura 3.2 Árvore de derivação para "real id,id,id"

3.1 Definição de Gramáticas de Atributos

Considere Δ e Σ como dois alfabetos finitos. Uma GA é uma 4-upla (G,A,VAL,SF). **G** é uma gramática livre do contexto (N,Σ,S,P), onde **N** é um conjunto de Não-Terminais, Σ é um conjunto de Terminais, **S** é o Símbolo Inicial, e **P** é um conjunto de Produções. **A** é um conjunto de atributos. Cada atributo **a** $\in$ **A**, é da forma X.a, indicando que **a** é um atributo associado com o Não-Terminal **X**. Cada atributo **a** é também marcado como *herdado* ou *sintetizado*. VAL é uma função que mapeia cada atributo **a** em **A** para um conjunto de possíveis valores VAL(a).

SF é um conjunto de *funções semânticas*, onde f $\in$ SF é associada com alguma produção p $\in$ P. Se f $\in$ SF está associada com [p: X_0 ::= $a_0X_1a_1X_2$ ...$a_{np}X_{np}$], então f é da forma: $X_{i1}.a_1 = g(X_{i1}.a_2,...,X_{ir}.a_r)$, onde g é um mapeamento de $VAL(X_{i2}.a_2)x...xVAL(X_{ir}.a_r)$ para $VAL(X_{i1}.a_1)$, ij(1<=j<=r) esta em [1..np], e cada $X_{ij}.a_{1j}$ está no conjunto de atributos de A. Existe exatamente uma função semântica em SF para computar cada atributo sintetizado de X_0 em p e cada atributo herdado de X_i(1<=i<=np) em p [YEL88].

4. TRANSFORMAÇÕES ENTRE LINGUAGENS

A conversão automática entre notações e entre linguagens de programação é uma técnica importante para aumentar a reusabilidade de programas. Muitas vezes existe a necessidade de transportar todo um sistema para um ambiente operacional cuja linguagem básica é outra. Outras vezes deseja-se utilizar as vantagens de uma nova linguagem sem perder os programas já escritos na linguagem anterior.

A tradução de um programa entre duas linguagens deve garantir a equivalência semântica dos programas traduzidos. Além disso, é essencial que a tradução preserve tanto quanto possível a estrutura do programa original. Preservando a estrutura aumenta-se o grau de legibilidade, entendimento e facilita-se a manutenção do programa gerado. O programador, ao estruturar seu programa, implicitamente constrói um modelo da solução e do domínio do problema. Destruindo-se a estrutura do programa perde esse modelo que possibilitaria, caso preservado, uma maior compressão das decisões de projeto que guiaram à sua construção.

Algumas linguagens, embora não possam ser consideradas equivalentes, apresentam visões diferentes de um mesmo objeto. Este caso é particularmente importante em ambientes que desejam fornecer ao usuário ferramentas que permitam a conversão automática entre diversos documentos. Como exemplo, pode-se citar os Ambientes de Desenvolvimento de Software [CAS90].

Uma maneira de converter uma linguagem em outra é a utilização de uma linguagem canônica.

Se uma representação equivalente de um programa em uma linguagem L para outra linguagem L' é desejada, então pode-se ter um passo intermediário, ou seja, converter L para uma linguagem canônica e depois converter da linguagem canônica para a linguagem L'. O motivo desta opção é a tentativa de diminuir esforço para mapear **n** linguagens em **m** linguagens, que exigiria a construção de **m.n** tradutores se não for utilizada uma linguagem canônica. Com o uso de linguagem canônica tem-se **m+n** tradutores.

A linguagem canônica deve ser *adequada* o suficiente para representar todas as linguagens de programação. Segundo [YEL88] esta é considerada *adequada,* se:

1. Todo programa pode ser expresso como um programa na forma canônica; e

2. Todo programa na forma canônica que é a imagem de uma tradução deve ser expresso como a imagem de todas as outras traduções.

Em [YEL88] é proposta uma linguagem canônica a linguagem Pascal e para a linguagem C. Para tal, é feito um levantamento sobre as características e diferenças necessárias para criação desta linguagem canônica.

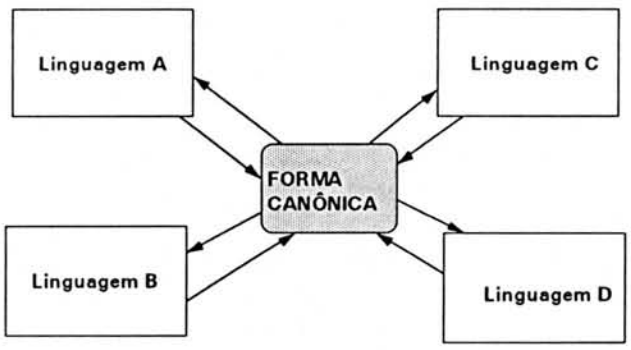


Figura 4.1 Linguagem canônica

4.1 Definição de Gramática Transformacional com Atributos

Uma das formas de descrever transformações entre linguagens é a utilização das Gramáticas Transformacionais com Atributos (GTAs). Para tal, basta o usuário descrever a linguagem 1 e as transformações para a linguagem 2.

Assim, pode-se definir uma GTA como uma 8-upla $T=(N,\Sigma,\Delta,R,S,A,VAL,SF)$, onde:

(1) N é um conjunto finito de símbolos não-terminais;
(2) Σ é um alfabeto finito de entrada;
(3) Δ é um alfabeto finito de saída;
(4) R é um conjunto finito de regras da forma
P ->α,β, onde $\alpha \in (N \cup \Sigma)^*$, $\beta \in (N \cup \Delta)^*$.
(5) S é um não-terminal de N
(6) A é um conjunto de atributos. Cada atributo **a** _$\in$ **A**, é da forma X.a, indicando que a é um atributo associado com o Não-Terminal **X**. Cada atributo **a** é também marcado como *herdado* ou *sintetizado*.
(7) VAL é uma função que mapeia cada atributo **a** em **A** para um conjunto de possíveis valores VAL(a).
(8) SF é um conjunto de *funções semânticas*, onde f $\in$ SF é associada com alguma produção p $\in$ P. Se f $\in$ SF está associada com [p: X_0 ::= $a_0X_1a_1X_2$ ...$a_{np}X_{np}$], então f é da forma: $X_{i1}.a_1 = g(X_{i1}.a_2,...,X_{ir}.a_r)$, onde g é um mapeamento de $VAL(X_{i2}.a_2)x...xVAL(X_{ir}.a_r)$ para $VAL(X_{i1}.a_1)$, ij(1<=j<=r) esta em [1..np], e cada $X_{ij}.a_{1j}$ está no conjunto de atributos de A. Existe exatamente uma função semântica em SF para computar cada atributo sintetizado de X_0 em p e cada atributo herdado de X_i(1<=i<=np) em p [YEL88].

5. FERRAMENTAS TRANSFORMACIONAIS ESTUDADAS

Pode-se descrever gramáticas transformacionais a partir de ferramentas já existentes, porém seu uso é dificultado, pois para tal deve-se escrever código (em linguagem de programação) para fazer as regras de transformação. Dentre as ferramentas existentes, diversas se baseiam em gramáticas de atributos para fazer as transformações. Em [PIE88] são apresentadas diversas ferramentas, com as quais é possível realizar transformações. [YEL88] vai mais longe e propõe uma ferramenta para tradução bi-direcional usando inversão de gramática de atributos.

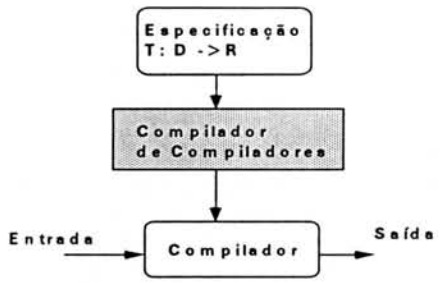


Figura 5.1 Compilador de Compiladores

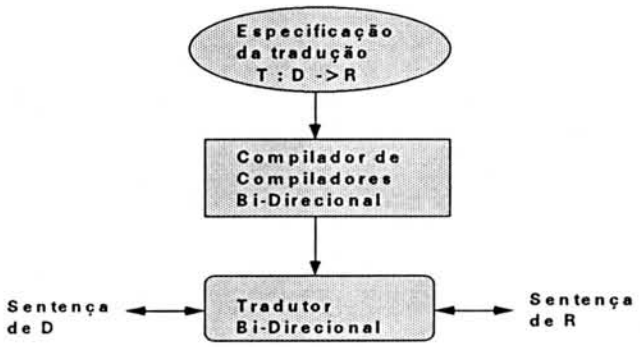


Figura 5.2 Compilador de Compiladores Bi-Direcional

As ferramentas analisadas e exemplificadas neste Capítulo referem-se a ferramentas desenvolvidas no Instituto de Informática e Curso de Pós-Graduação em Ciência da Computação da Universidade Federal do Rio Grande do Sul, SINLEX [ROS89] e GG [QUA91]; e a uma ferramenta muito utilizada por diversos pesquisadores, YACC [JOH75].

5.1 YACC - Yet Another Compiler Compiler

O YACC é um utilitário do sistema UNIX [JOH75], usado no desenvolvimento de processadores de linguagens, que gera, a partir de uma descrição, um analisador sintático ascendente. Este analisador, ou "parser", reconhece sentenças que pertençam a uma linguagem descrita por uma Gramática Livre do Contexto [AHO86] (GLC) especificada em BNF, onde o usuário especifica um conjunto de produções que definem as possíveis construções da linguagem de entrada e as ações, associadas a cada produção, a serem realizadas toda vez que estas produções forem reconhecidas.

O YACC é um gerador de tabelas do tipo LALR(1), para executar um algoritmo de análise LR (*Left-to-right rightmost derivation in Reverse*) [AHO86]. O *driver* de reconhecimento é constante, sendo gerado somente o conjunto de tabelas que contém a descrição de uma determinada linguagem.

Abaixo segue um exemplo de descrição de produção do YACC, sem a utilização de ações semânticas.

```
STATEMENT : 'if' BOLLEXPR 'then' STATEMENT
     | 'if' BOLLEXPR 'then' STATEMENT 'else' STATEMENT
     | 'while' BOOLEXPR 'do' STATEMENT
     | 'begin' STATEMENTLIST 'end'      | ;
```

5.1.1 Ações Semânticas

Para cada regra da gramática, o usuário pode associar ações a serem realizadas toda vez que a regra for reconhecida no processo de entrada. Estas ações podem retornar valores, e receber valores retornados por ações prévias, ou seja, cada ação semântica possui associado a si atributos.

Uma ação é um conjunto de comandos em linguagem de programação C.

As ações semânticas são executadas somente quando uma produção (regra) gramatical for reconhecida. Para permitir que ações semânticas possam ser inseridas no meio de uma produção e que se respeite a regra anterior, o YACC converte ações semânticas inseridas no meio da produção em novas produções que levam ao símbolo vazio. Desta forma, quando a produção que leva a vazio for reconhecida então a produção pode ser executada.

```
A : B { $$ = 1; } C { $$ = $2; };
```

Na ação semântica { $$ = 1; } quem está recebendo o valor 1 é o atributo associado à ação semântica e não ao atributo do não-terminal A. O valor do atributo A só é calculado em { $$ = $2; }. Isto se deve as ações semânticas só serem executadas no momento do reconhecimento de uma produção. Para contornar este problema o YACC cria uma nova produção que possui somente a ação semântica que está no meio da produção. Veja abaixo como é convertida a produção acima.

```
NOVOA : { $$ = 1; } ;

A : B NOVOA C { $$ = $2; };
```

5.1.2 Atributos

Cada símbolo de produção possui associado uma pseudo-variável, cujo nome inicia com "$" e é complementado com um índice que indica a qual símbolo esta pseudo-variável é associada.
Assim,

```
A : B C { $$ = $1 + $2; }
```

a pseudo-variável "$$" é associada com o não-terminal "A", $1 é associada com o símbolo B e $2 é associada com o símbolo C. Os símbolos B e C podem ser não-terminais ou terminais ou, ainda, ações semânticas. Os índices são numerados da esquerda para a direita, sendo o símbolo $$ sempre (que a ação semântica está no fim da produção) associado ao não-terminal que deriva na produção em questão.

Por *default*, os valores que podem ser armazenados nas pseudo-variáveis são do tipo *int* (inteiro). O YACC pode suportar outros tipos, inclusive estruturas como registros. Com o YACC o usuário pode ainda escolher mais de um tipo para estas pseudo-variáveis [SUN90].

Uma das formas de definir um novo tipo para os símbolos da gramática é através da redefinição do tipo YYSTYPE.

```
/**  Definição do tipo dos atributos de cada Símbolo
     da gramática. YYSTYPE é o tipo dos atributos.
**/
%{
typedef struct {
          int val;
          int pos;
              } YYSTYPE;
%}

/**
     Início da definição da gramática
**/

%%

NROBIN : LBIN { printf("Decimal=%d\n",$1.val); } ;

LBIN : DIG LBIN
          { $$.val = $2.val + $1.val * POT(2,$2.pos);
            $$.pos = $1.pos + 1;
           }
       |  { $$.val = 0; $$.pos = 0; } ;

DIG : 0  { $$.val = 0; }
      | 1 { $$.val = 1; }  ;

%%
```

Gramática 5.1 Conversão de binário para decimal (1)

A gramática aceita como entrada qualquer número binário. Durante o reconhecimento é feita a conversão de Binário para Decimal. Neste exemplo existe recursividade a direita e utilização de atributos sintetizados (A Figura 6.3 apresenta um exemplo utilizando recursividade a esquerda e com atributos herdados e sintetizados).

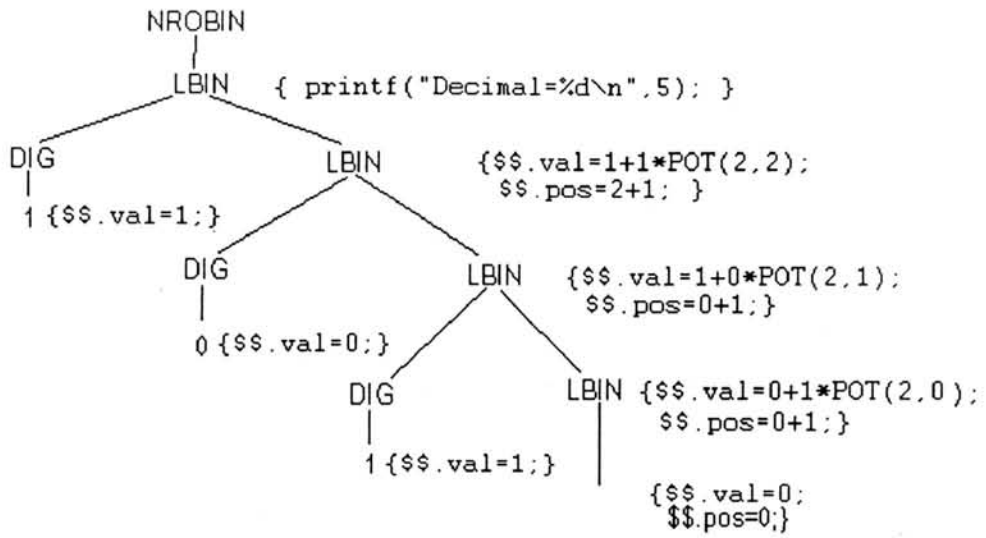


Figura 5.3 Árvore de derivação para "101"

Tabela 5.1 Reconhecimento da entrada "101"

ENTRADA	PILHA SÍMBOLOS	REDUÇÃO	AÇÃO	PILHA VALOR
101	-	-	-	-
01	1	-	-	-
01	DIG	DIG:1	{ $$.val = 1; }	1
1	DIG 0	-	-	1
1	DIG DIG	DIG:0	{ $$.val = 1; }	1 1
-	DIG DIG 1	-	-	1 1
-	DIG DIG DIG	DIG:1	{ $$.val = 1; }	1 1 1
-	DIG DIG DIG LBIN	LBIN:ε	{ $$.val = 0; $$.pos = 0; }	1 1 1 (0,0)
-	DIG DIG LBIN	LBIN : DIG LBIN	{ $$.val = 1; $$.pos = 1; }	1 1 (1,1)
-	DIG LBIN	LBIN : DIG LBIN	{ $$.val = 1; $$.pos = 2; }	1 (1,2)
-	LBIN	LBIN : DIG LBIN	{ $$.val = 5; $$.pos = 3; }	(5,3)
-	NROBIN	NROBIN : LBIN	{ printf(5); }	-

5.1.3 Atributos Herdados

A utilização de atributos no YACC é simples quando usam-se atributos sintetizados (ver exemplo anterior). Quando é necessário o uso de atributos herdados, a complexidade de utilização fica um pouco maior.

Uma ação pode referenciar atributos calculados por uma ação anterior. O mecanismo de utilização é tão simples quanto os anteriores, ou seja um "$" seguido por um índice, só que neste caso o índice deve ser 0 (zero) ou negativo.

Considere a gramática com as ações abaixo, extraída de [SUN90]:

```
SENT : ADJ NOUM VERB ADJ NOUM ;

ADJ : "the"   { $$ = "the"; }
    | "young"  { $$ = "young"; }
    .....
    ;

NOUM  :  "dog"   { $$ = "dog"; }
    | "crone" {  if (strcmp($0,"young"))
                        printf ....
              }
```

Na ação que segue a palavra "crone", é feito uma comparação para verificar se a palavra reconhecida imediatamente anterior foi "young". Isto só é possível pois na estrutura da gramática acima sabe-se que antes de NOUM vem um ADJ. Se uma outra produção possuísse o símbolo NOUM fazendo parte de sua produção então poder-se-ia ter alguns problemas derivados deste fato.

```
PROD : NTERM NOUM;
```

Se a produção acima estivesse inserida na gramática apresentada então a ação semântica que se refere ao atributo $0 não mais estaria referenciando o atributo do não-terminal ADJ, mas estaria se referindo ao atributo do não-terminal NTERM.

Outra característica importante, que NÃO é suportada pelo YACC, é a herança de atributos independente de um símbolo já ter sido reconhecido ou não. No exemplo abaixo, o valor do atributo de TIPO é necessário antes de reconhecer a LISTA de identificadores (Ver exemplo abaixo).

```
(1)  DECL : { $2.type = $4.type; } LISTA ":" TIPO ;
(2)  TIPO  : "int" { $$.type = int; };
(3)  LISTA : LISTA "," IDENT | IDENT ;
```

A ação semântica apresentada em (1) está incorreta, pois LISTA é avaliada antes de TIPO, e portanto, o atributo de TIPO ainda não existe. Além deste fato, o YACC não suporta este tipo de ação semântica , devido ao problema apresentado em 5.1.1.

5.1.4 Transformações

A inserção de regras de transformação no YACC é possível, mas com um custo elevado. Para realização de regras de transformação o usuário deve utilizar-se de comandos em linguagem C que realizem as regras, sendo algumas vezes uma tarefa muito cansativa, inclusive com criação de novas estruturas de dados.

Abaixo é apresentado um exemplo de transformação de um comando da linguagem Pascal para a linguagem C.

```
(1)  STAT  : "for" VAR ":=" EXPR "to" EXPR "do" STAT
           {
                printf("for(%s=%s;%s<=%s;%s++)\n",
                     $2.str,$4.str,$6.str,$2.str);
           }

(2)  EXPR : EXPR "+" EXPR
           { strcpy($$.str,$1.str);
             strcat($$.str,"+");
             strcat($$.str,$3.str);
           }
(3)        | EXPR "*" EXPR
           { strcpy($$.str,$1.str);
             strcat($$.str,"*");
             strcat($$.str,$3.str);
           }
(4)        | IDENT { strcpy($$.str,$1.str); }
           | ...
```

A operação *strcpy(s1,s2)* copia *s2* para *s1*; *strcat(s1,s2)* concatena em *s1* a string *s2* . Exemplo: se *s1* tem o valor "AB" e *s2* tem o valor "CD" então após a operação *strcat(s1,s2)* *s1* terá o valor "ABCD".

5.2 SINLEX - Ambiente de Desenvolvimento de Processadores de Linguagens

O sistema SINLEX é um Ambiente de Desenvolvimento de Processadores de Linguagens, sendo formado por um tradutor da Linguagem de Descrição de Gramáticas (LDG)

integrado a um editor de texto e por uma interface de teste e depuração da gramática que simula o analisador a ser gerado pelo sistema. As gramáticas aceitas pelo SINLEX são as LL(1), sendo esta a grande desvantagem da utilização do sistema. A descrição da gramática é feita em uma notação do tipo BNF. O reconhecedor é gerado na forma descendente recursivo em linguagem C ou Pascal.

5.2.1 Atributos

O sistema SINLEX utiliza um esquema de tradução dirigido pela sintaxe através de gramática de atributos incorporada na LDG. É permitida aos Não-Terminais da gramática, a declaração e uso de parâmetros e variáveis locais. Também é permitida a utilização de variáveis globais a quaisquer Não-Terminais.

NT (*parâmetros*) -> LOCAL (*variáveis locais*) *corpo da produção*

É permitida, também, a inclusão de ações semânticas em qualquer lugar do corpo das produções da gramática.

NT -> S1 { *ação semântica* } S2 ....

No sistema SINLEX pode-se ter atributos herdados ou sintetizados, já que existe a possibilidade de passar os atributos como se fossem parâmetros de rotinas da linguagem na qual será gerado o analisador descendente recursivo. Os atributos herdados possuem o inconveniente, em algumas gramáticas, de serem analisados antes para depois serem utilizados, ou seja, podem ser herdados do seu pai e dos seus irmãos da esquerda. No exemplo abaixo,

para passar o valor do TIPO para os identificadores da LISTA não é possível pois o identificador de tipo só é reconhecido após o reconhecimento de toda a lista de identificadores.

```
(1)  DECL  -> LOCAL (Var ta : TipoAtributo;)
                 LISTA(ta) ":" TIPO(ta) ;
(2)  TIPO(Var ta:TipoAtributo) -> "int"
                                  { ta = int; };
(3)  LISTA(ta:TipAtributo) -> LISTA(ta) "," IDENT
                            | IDENT ;
```

O parâmetro (ta - atributo de tipo) passado para LISTA não tem sentido, pois LISTA é avaliada antes de TIPO, e portanto, o atributo (ta) de TIPO ainda não foi calculado.

No exemplo anterior, poder-se-ia resolver o problema utilizando atributos sintetizados, mas o custo poderia ser mais elevado.

5.2.2 Transformações

Assim como no YACC, a possibilidade de transformações no SINLEX existe, mas com um custo elevado. Para fazer conversões existe a necessidade de programar tais transformações e dependendo da complexidade da gramática, esta programação pode ser muito custosa.

Abaixo encontra-se um exemplo de conversão do comando "FOR" da linguagem Pascal para o comando "FOR" na linguagem C. É apresentado também o início das regras necessárias para a conversão de expressões escritas em

Pascal para expressões escritas em C (na Seção 5.1.4 foi apresentado o mesmo exemplo para o YACC).

```
STAT -> "for" VAR (V) ":=" EXPR (E1) "to" EXPR (E2)
        "do" STAT
        {
     Writeln("for (",V,"=",E1,",",V,"<=",E2,",",V,"++)");
        }

EXPR (E:String) ->  EXPR (E1) "+" EXPR(E2)
                    { E := E1 + '+' + E2; }
          |     EXPR (E1) "*" EXPR(E2)
                { E := E1 + '*' + E2; }
          | IDENT(I) { E := I; }
          | ....
```

5.3 GG - Gerador Automático de Código Objeto usando Gramáticas Transformacionais

O sistema GG utiliza gramáticas transformacionais para automatizar o processo de geração de código objeto. Utilizando as GTs, é apresentada uma metalinguagem de compilação cuja função é a de permitir a descrição do processo compilativo de maneira rápida e eficiente. Para este fim, as gramáticas transformacionais foram estendidas a fim de servir à área de compiladores. Assim, através de sua utilização, pode-se descrever como um programa fonte é "transformado" em um programa objeto. A metalinguagem apresentada é do tipo BNF, sendo que a parte de transformações informada ao fim de cada produção.

O processo de transformação de programas fonte em programas objeto é dissociado da análise sintática, sendo

feito como uma fase posterior a esta. Esta transformação é realizada sobre a árvore de derivação do programa fonte, resultando uma árvore de derivação transformada que representa o programa objeto, sendo o código gerado obtido das suas folhas.

O sistema é composto de duas partes: uma responsável pela geração automática de código objeto e que foi denominada GAGO; e outra responsável pelas fases precedentes à geração de código dentro do processo compilativo, ou seja, os geradores de analisadores léxicos e sintáticos. Para a segunda parte, foi utilizado o sistema SINLEX, tendo este sido modificado para integrar o sistema GG. Do SINLEX foram retiradas as partes responsáveis pela execução de ações semânticas e pela gramática de atributos.

5.3.1 LDGT - Linguagem de Descrição de Gramáticas Transformacionais

A GT é descrita na forma de listas de transformações associadas às produções sintáticas. Assim, cada produção da linguagem fonte tem uma lista de transformações associada. Esta lista de transformações descreve as operações, ou transformações, necessárias para mapear a produção fonte em uma produção objeto. Uma lista de transformações é um conjunto de itens transformacionais, os quais podem ser de um entre 5 diferentes tipos, que realizam funções distintas no processo transformacional. São eles *índice, código, rotinas semânticas, parâmetro* e *predicado condicional*.

A LDGT utiliza uma forma semelhante a BNF para a definição das regras léxicas e sintáticas. Uma especificação de gramática em LDGT divide-se em três

seções (Léxica, de Tokens e Sintática). A seguir tem-se um esqueleto desta especificação:

```
Seção Léxica
      Conjuntos
      Regras Léxicas
Seção de Tokens
      Delimitadores
      Tokens e Listas de Palavras Reservadas
Seção Sintática e Transformacional
```

A seção sintática e transformacional constitui-se de regras ou produções transformacionais. Cada produção transformacional pode ser dividida em três partes distintas como segue:

```
NÃOTERMINAL -> CORPO : TRANSFORMAÇÃO
```

Os tipos de itens que compõem a parte transformacional são separados por vírgulas e possuem o seguinte significado:

-índice: é um número que referencia um elemento do corpo da produção. Se o elemento for um não terminal, então lhe é permitido passar parâmetros. No exemplo abaixo o símbolo 3 da produção é passado como parâmetro para o símbolo 2, na regra de transformação.

```
NT -> "t1" NT1 "t2" NT2 : 1, 2(3), 4.
```

-código: é uma cadeia de caracteres que não pode ter parâmetros associados. No exemplo abaixo o símbolo "p1" que aparece na regra de transformação é *código*.

```
NT -> "t1" NT1 "t2" NT2 : 1, "p1" ,4.
```

-parâmetro: é o delimitador % seguido de um número que referencia um parâmetro passado.

```
NT -> NT1 NT2 : %1, %2, 1.
```

-rotina semântica: é um identificador que referencia uma rotina escrita em Pascal.

```
NT -> NT1 "t1" : Rotina1, Rotina2(2, "p1").
```

-predicado condicional: serve para executar uma entre duas transformações possíveis.

No exemplo abaixo, é apresentada uma gramática para converter uma sentença da voz ativa do português para a voz passiva do português.

```
FRASE -> "o" SUJEITO "comeu" "o" SUJEITO :
          4,5,"foi comido pelo",2.
SUJEITO -> "cão" | "gato" | "rato" : 1
```

O índice 4 se refere ao artigo "o", o índice 5 se refere ao segundo SUJEITO, o índice 2 se refere ao primeiro SUJEITO. Desta forma, a frase: "o gato comeu o rato" será transformada para "o rato foi comido pelo gato".

5.4 Resumo Comparativo

A seguir são listadas as vantagens e desvantagens dos sistemas apreciados:

-YACC: Gramática de entrada potente, LALR(1), analisadores gerados com bom desempenho, boa gramática de atributos, dificuldade de construir regras

transformacionais;

-SINLEX: Gramática de entrada pobre, ELL(1), analisadores gerados com ótimo desempenho, gramática de atributos boa, dificuldade de construir regras transformacionais;

-GG: Gramática de entrada pobre, ELL(1), analisadores gerados com ótimo desempenho, não possui gramática de atributos, ótima técnica para construir regras transformacionais, muito lento para gerar a árvore transformada.

A seguir é apresentada uma ferramenta que se utiliza de Gramáticas Transformacionais com Atributos, que tenta agregar as características boas das ferramentas anteriores, e corrigir os problemas nelas encontrados.

6. FERRAMENTA PROPOSTA

Um grande conjunto de programas de computador tem como entrada sentenças que possuem alguma estrutura, e tais programas convertem estas sentenças para novas sentenças com uma estrutura diferente. As sentenças de entrada respeitam uma "linguagem de entrada" que é aceita pelo programa. Esta linguagem de entrada pode ser tão complexa quanto uma linguagem de programação, ou tão simples como uma sequência de números. A conversão para a "linguagem de saída" é uma tarefa pesada, se levarmos em conta a complexidade da linguagem de entrada ou da própria linguagem de saída.

O UCC - *Um Compilador de Compiladores* fornece uma ferramenta geral para descrever a entrada para programas de computador e como tais entradas podem ser convertidas em uma saída desejada. O usuário do UCC especifica a estrutura da entrada, junto com trechos de código (ações semânticas) que devem ser executados no momento que cada item da linguagem é reconhecido. Para cada item da estrutura de entrada o usuário pode especificar uma regra de transformação para uma linguagem de saída. O UCC transforma estas especificações em uma descrição para o YACC, que por sua vez, gera uma rotina para manipular o processo de entrada.

Juntamente com a rotina gerada pelo YACC, são anexadas pelo UCC, rotinas para conversão da linguagem de entrada (vale salientar, reconhecida pela rotina gerada pelo YACC) para a linguagem de saída.

A classe de gramáticas aceitas pelo UCC, tanto para a entrada como para a saída, é uma das mais gerais: gramáticas LALR(1), sem ambigüidades.

6.1 Estrutura da ferramenta

A entrada do UCC é a Linguagem de Descrição de Gramáticas Transformacionais com Atributos que será descrita na Seção 6.2. A descrição é submetida ao UCC que verifica a consistência da descrição. Eventuais erros são armazenados em um arquivo e o usuário é notificado.

Após o usuário ter submetido o arquivo de entrada, que contém a LDGTA, o UCC está apto a fazer uma das atividades abaixo:

- gerar um *parser* para a Linguagem 1;

- gerar um *parser* para a Linguagem 2;

- gerar um tradutor da Linguagem 1 para a Linguagem 2;

- gerar um tradutor da Linguagem 2 para a Linguagem 1.

Após esta escolha, o UCC gera uma arquivo de descrição de uma gramática para o YACC. O YACC recebe este arquivo e gera o parser para realizar a operação escolhida pelo usuário (uma das quatro anteriores).

Concluídas estas etapas, a seguir é feita a compilação das rotinas do projetista com as rotinas do sistema responsáveis pela construção da árvore de derivação e das transformações das linguagens.

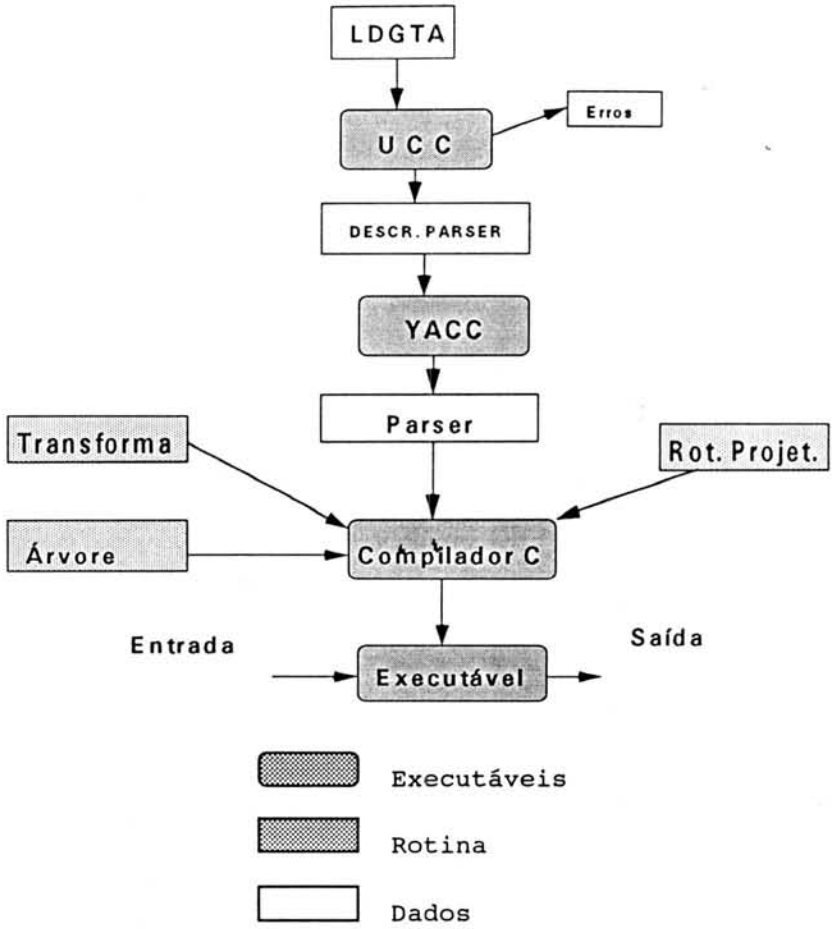


Figura 6.1 Estrutura da ferramenta transformacional

6.2 Especificação do arquivo de entrada

Cada arquivo de especificação para o UCC é dividido em três seções: *declarações*, *regras das gramáticas*, e *programas*. As seções são separadas por %%. A seção de declarações e programas seguem as características da ferramenta YACC, para tanto, deve-se consultar [SUN90].

A estrutura do arquivo é a seguinte:

declarações
%%
regras
%%
programas

As seções de declarações e programas podem ser omitidas, sendo deste modo, a menor especificação para um arquivo do UCC:

%%
regras

Os caracteres *space, tabs, carriage return* e *line feed* não são considerados, exceto quando aparecem na composição de nomes. Comentários podem aparecer em qualquer parte do texto e são iguais ao usado na linguagem de programação C.

```
/* ..... comentário ......*/
```

A seção de regras é composta por uma ou mais regras gramaticais. Uma regra gramatical possui o seguinte formato:

```
A : CORPO1 -> CORPO2 ;
```

Onde, A representa um *não-terminal*, CORPO1 e CORPO2 representam uma seqüência de zero ou mais símbolos das gramáticas. Esta seqüência de símbolos especifica a linguagem de entrada e a linguagem de saída. O símbolo ":" representa que o não-terminal A deriva em CORPO1 ou CORPO2. O símbolo "->" serve para separar as duas gramáticas.

Os símbolos podem ter tamanho ilimitado, e devem ser compostos de letras, sublinhado (_) e dígitos (desde que estes não iniciem um símbolo). Letras maiúsculas e minúsculas são consideradas diferentes. Os símbolos podem ser terminais ou não-terminais.

Um terminal literal consiste de um ou mais caracteres delimitados por apóstrofo ('). Como na linguagem C, a barra invertida (\) é um caractere especial dentro de literais, e todos os caracteres especiais de C são reconhecidos:

```
'\n' newline
'\r' carriage return
'\'' aspas simples (')
'\\' barra invertida
'\t' tabulação
'\b' backspace
'\f' form feed
'\xxx'    xxx é um número octal
```

Devido a restrições técnicas da ferramenta YACC, o caractere '\0' nunca deve ser usado.

Se existe um conjunto de regras gramaticais para o mesmo não-terminal, a barra vertical pode ser usada para reescrever as regras. O ponto-e-vírgula (;) no final representa o término das regras para aquele não terminal.

Assim, as regras:

```
A : B C D -> D C B ;
A : E F -> F E;
A : G -> G ;
```

Podem ser reescritas como:

```
A : B C D -> D C B
    | E F -> F E
    | G -> G
    ;
```

Não é necessário que todas as regras gramaticais com

o mesmo não-terminal apareçam juntas na seção de regras, embora esta característica torne a descrição da entrada mais clara e fácil de ser alterada.

Se um não-terminal deriva uma seqüência vazia de símbolos, então a regra gramatical pode ser escrita como:

```
A : -> ;
```

A especificação da regra que descreve a saída nem sempre é necessária e portanto pode ser suprimida. Desta forma a descrição da regra gramatical pode ser:

```
A : CORPO ;
```

Durante a fase de conversão de uma linguagem para outra, a produção que possuir este formato é desconsiderada.

Os símbolos que aparecem nas regras gramaticais, que não são terminais, devem aparecer pelo menos uma vez do lado esquerdo de uma regra gramatical.

No corpo das produções podem aparecer cinco (5) tipos de elementos:

-terminais: aqueles que não derivam em nenhuma produção. Podem ser: um literal, ou seja uma cadeia de caracteres delimitada por aspas (") (ex: "aaa"); ou um terminal reconhecido pelo analisador léxico (ex: num).

-não-terminais: aqueles que são lado esquerdo de produção. Neste caso existe uma restrição. Os não-terminais que aparecem na parte transformacional devem aparecer também no corpo da produção. Existe uma possibilidade para eles não aparecerem no corpo da produção, quando estes não-terminais podem levar a uma

produção vazia na parte transformacional. Veja exemplo abaixo.

```
A : B C A -> B C D A
    | ;
B :  "b"  -> "B"
    | "B" -> "b"
C :  "c"  -> "C"
    | "C" -> "c";
D :  "d"  -> "D"
    | "D" -> "d";
```

Gramática 6.1 Troca "B" <-> "b", e "C" <-> "c"

A gramática anterior recebe uma seqüência de Bs e Cs e transforma para o seu inverso seguido de um D, ou seja, se vem um "B" (maiúsculo) então ele é substituído por um "b" (minúsculo). O mesmo acontece com o C (para a entrada "bCbc" será gerado "BcBC"). O problema aparece no momento de escolher a produção que D deriva na parte transformacional: a primeira produção ou a segunda? Não existe forma de determinar, sem o auxílio do usuário, e esta atitude não é desejável. Para evitar este tipo de situação uma produção do tipo D : -> ; deveria aparecer.

-rotinas: o usuário pode desejar fazer alguma modificação em um terminal antes de fazer uma transformação. Para isto, existe a possibilidade de o mesmo utilizar uma rotina que faz tal conversão. Abaixo está a sintaxe da utilização de uma rotina sobre um terminal.

```
Não-terminal : ... @nome_rot(terminal)  ... -> ...;
```

Um exemplo prático seria a conversão de *strings* da linguagem Pascal para *strings* da linguagem C. Em Pascal uma *string* é delimitada por apóstrofos (') enquanto que em C é delimitada por aspas (").

```
NT :  ...@converte(STRING)... -> ...@converte(STRING)...
```

As rotinas são executadas somente na parte transformacional, não sendo consideradas no momento do reconhecimento de uma sentença de entrada.

-atributos: o valor dos atributos dos símbolos da gramática também podem ser colocados na regra de transformação. Para este aspecto, o usuário deve especificar qual atributo de qual símbolo está sendo utilizado. Os atributos só são considerados no momento da transformação, durante o reconhecimento de uma sentença de entrada este símbolo não é considerado.

```
A : ... -> ... $1.val ...;
```

não-terminais com índice: outra situação que deve ser considerada é a de possuir um símbolo repetido em uma produção. Como identificar no momento da transformação qual deve ser utilizado.

```
E : E "+" E -> E E "+";
```

Na produção acima, deve-se ter um modo de identificar em que ordem os não-terminais E aparecem na regra transformacional acima. Se tivermos a ordem apresentada acima, então o primeiro E que aparece na regra transformacional corresponde ao primeiro E da produção da gramática de entrada. Se o usuário deseja inverter a ordem, então ele deve especificar através de um índice.

```
E -> E "+" E -> E#2 E#1 "+";
```

O símbolo "#" serve para identificar qual dos não-terminais está sendo utilizado.

6.3 Ações Semânticas

À cada regra gramatical da linguagem de entrada, o usuário pode associar ações que serão executadas quando a regra é reconhecida no processo de entrada, ou após toda a entrada ter sido reconhecida.

Uma ação é um trecho de programa escrito na linguagem C e, como tal, pode fazer entrada e saída, chamada a sub-rotinas, alterar vetores externos e variáveis. Uma ação é delimitada por chaves '{' '}' ou colchetes '[' ']'.

Existem dois tipos de ações semânticas associadas com cada produção. As ações semânticas BOTTOM-UP e as ações semânticas TOP-DOWN.

6.3.1 Ações BOTTOM-UP

Estas ações são delimitadas por chaves { } e são executadas durante o processo de reconhecimento da entrada.

Por exemplo:

```
A : '(' B ')'
{
 printf("Reduz a produção ( B ) para o símbolo A");
 };
```

Este tipo de ação semântica é executado somente no momento em que uma produção é reduzida para um não-terminal. Portanto, produções como:

A : B { *ação 1* } C { *ação 2* } ;

nas quais as ações semânticas são inseridas no meio da produção, são transformadas para duas novas produções da seguinte forma: é criada uma nova regra com um novo símbolo não-terminal, sendo que esta regra deriva a produção vazia, e assim após fazer a redução para este novo símbolo, a ação semântica é executada. Este mecanismo torna a utilização de atributos um pouco complexa como foi mostrado na Seção 5.1.1.

Novo : { *ação 1* } ;
A : B *Novo* C { *ação 2* } ;

6.3.2 Ação TOP-DOWN

Estas ações são delimitadas por colchetes [ ] e são executadas após toda a entrada ter sido reconhecida e a árvore de derivação ter sido construída.

Exemplo:

```
A : [ printf("Não-terminal A deriva ( B ) "); ]
      '(' B ')'
```

A ação semântica acima é executada antes de se avaliar as ações semânticas que porventura venham a ser executadas a partir do símbolo B.

O problema gerado pelas ações semânticas BOTTOM-UP não aparece neste tipo de ações.

6.4 Atributos

Com a utilização do UCC o usuário pode especificar dois tipos de atributos: herdados e sintetizados. Para trabalhar com estes atributos o usuário deve utilizar as

ações semânticas vistas na Seção 6.3.

Por *default* os valores dos atributos são inteiros, mas o UCC suporta também atributos de outros tipos, incluindo estruturas, do mesmo modo que apresentado no YACC (ver Seção 5.1.2).

6.4.1 Atributos sintetizados usando ações BOTTOM-UP

Uma forma de trabalhar com atributos sintetizados é através das ações semânticas BOTTOM-UP. Conforme exemplo abaixo:

```
E : E '+' E { $$ = $1 + $3; } -> E#1 E#2 '+'
      | id    { $$ = $1; }    -> id;
```

onde o valor do atributo do não-terminal E é sintetizado a partir dos valores dos atributos dos não-terminais do corpo da produção. No caso o atributo do não-terminal E é sintetizado a partir do atributo do primeiro símbolo da produção ($1) somado com o atributo do terceiro símbolo da produção ($3). Ou a partir do valor do atributo de "id".

6.4.2 Atributos herdados usando ações semânticas BOTTOM-UP

A utilização de atributos herdados usando ações semânticas BOTTOM-UP é feita do mesmo modo que no YACC. Uma ação pode referenciar atributos calculados por uma ação anterior. O mecanismo de utilização é tão simples quanto os utilizados para atributos sintetizados, ou seja, um "$" seguido por um índice, só que neste caso o índice deve ser 0 (zero) ou negativo.

Consideremos a gramática com as ações abaixo:

```
A :  B C A      -> A B C
     | "a"      -> "a"

B :  "b" { $$ = 1; }      -> "b"
     | "B" { $$ = 0; }    -> "B";

C:"c" {if($0==0) ERRO("Antes de 'c' deve vir 'b'");}->"c"
 |"C" {if($0==1) ERRO(Antes de 'C' deve vir 'B'");}->"C";
```

Gramática 6.2 Seqüência de B C terminado por A

A gramática acima reconhece qualquer entrada que possua uma sequência de "bc" ou "BC" terminadas por "a". Na produção C é especificado que quando vier um "c" ele só pode ser aceito se anteriormente veio um "b". Esta verificação é feita através do atributo herdado de B, e utilizado na ação semântica através do símbolo $0. Se o símbolo $0 (note que ele é correspondente ao símbolo $$ do não-terminal B) possuir o valor 0 então um "B" foi reconhecido anteriormente; se possuir o valor 1 então um "b" foi reconhecido.

6.4.3 Atributos herdados e sintetizados usando ações semânticas TOP-DOWN

Com ações semânticas TOP-DOWN, a utilização de atributos fica facilitada, principalmente de atributos herdados. No processo BOTTOM-UP as ações são executadas no momento do reconhecimento e, portanto alguns atributos que serão necessários não foram calculados ainda, tornando o trabalho do usuário mais complicado. Com ações TOP-DOWN, as ações são executadas após a árvore de

derivação já estar criada, e após as ações semânticas BOTTOM-UP já terem sido executadas. Desta forma, têm-se uma árvore de derivação com alguns atributos já calculados, o que torna a execução das ações TOP-DOWN extremamente simples para o entendimento do usuário.

```
NROBIN :  [ $1.pos = 0; ] LBIN
          [ printf("Decimal=%d\n",$1.val); ] -> LBIN;

LBIN :    [ $1.pos = $$.pos + 1; ] LBIN DIG
          [ $$.val = $1.val+$2.val*potencia(2,$$.pos); ]
          -> DIG LBIN
          |    [ $$.val = 0; ] -> ;

DIG : '0'  [ $$.val = 0 ]  -> '0'
       | 1 [ $$.val = 1 ] -> '1';
```

Gramática 6.3 Conversão de binário para decimal (2)

Acima tem-se uma gramática para cálculo de um número binário para sua respectiva forma em decimal. Neste exemplo pode-se notar a utilização de atributos herdados e sintetizados. Na produção NROBIN : LBIN, tem-se duas ações semânticas associadas, a primeira faz com que LBIN possua um atributo herdado, ou seja ao campo *pos* de LBIN é atribuído o valor 0 (zero); na segunda ação semântica é utilizado um atributo sintetizado do não-terminal LBIN, ou seja o campo *val* que o possui o valor, em decimal, do número binário. Na produção LBIN : LBIN DIG, também são utilizados atributos herdados e sintetizados.

A transformação realizada no exemplo acima irá inverter a sentença de entrada. Assim a sentença "10010" será convertida para "01001".

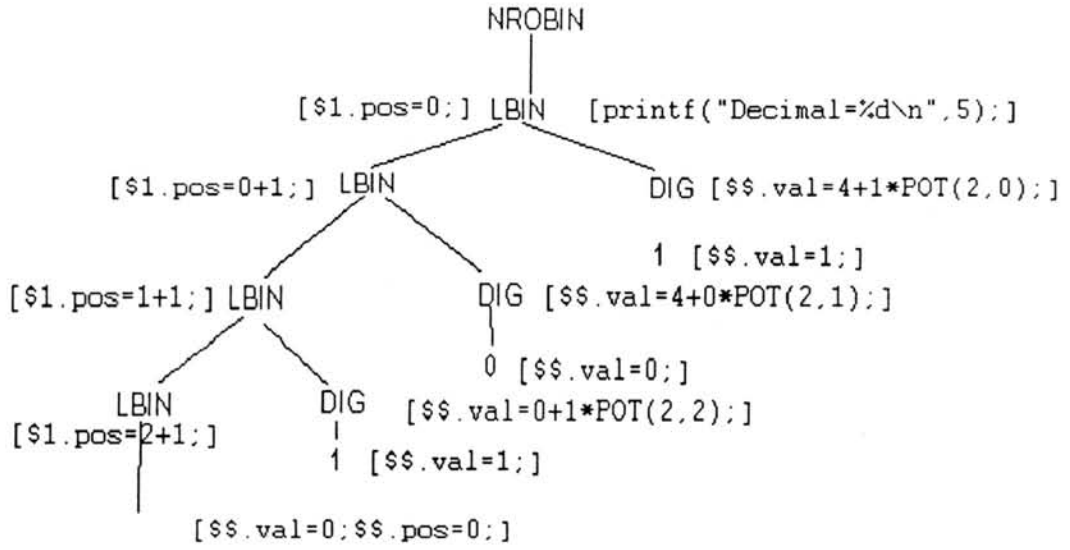


Figura 6.2 Árvore de derivação para "101"

6.5 Implementação

O UCC foi implementado em linguagem de programação C, no sistema operacional UNIX. Ele conta com aproximadamente 3.000 linhas de código fonte.

A linguagem de programação C foi escolhida por ser a mais utilizada pelos pesquisadores do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal do Rio Grande do Sul (CPGCC/UFRGS).

O *parser* gerado pela ferramenta é implementado em linguagem C, assim, pode ser feita uma fácil integração com outras ferramentas disponíveis no CPGCC/UFRGS. Como exemplo pode-se citar o gerador de analisadores léxicos LEX [SUN90].

O conjunto de rotinas básicas está escrito de uma maneira a poder ser transportado para qualquer ambiente de trabalho que suporte a linguagem C. A parte de

comunicação com o usuário é dependente do ambiente operacional que o mesmo estiver utilizando. Esta primeira versão suporta o ambiente das estações de trabalho SUN do CPGCC/UFRGS.

6.5.1 Estruturas de Dados

O **UCC** utiliza, internamente, um conjunto de listas encadeadas que armazenam os não-teminais, terminais, ações semânticas e produções de cada não-terminal. Estas listas são construídas no momento que o UCC está analisando o arquivo de entrada descrito pelo usuário. Veja a definição das estruturas abaixo. A definição está escrita na linguagem de programação C.

```
typedef struct tlistaprod {
     char *conteudo;
     int tipo;
     int indice;
     struct tlistaprod *prox;
} TListaProd;
```

A estrutura **TListaProd** é utilizada para armazenar as produções da gramática. No campo **conteúdo** é armazenado o símbolo da produção. No campo **tipo** está indicando se o símbolo da produção é um não-terminal, terminal, acão semântica TOP-DOWN ou ação semântica BOTTOM-UP. O campo **índice** representa, na parte tranformacional, qual dos não-terminais da produção deve ser utilizado.

```
typedef struct tproducoes {
     TListaProd *prod1;
     TListaProd *prod2;
     struct tproducoes *prox;
} TProducoes;
```

A estrutura **TProducoes** é utilizada para armazenar todas as produções de um determinado não-terminal. Como o UCC pode fazer transformação da linguagem 1 para a linguagem 2 ou vice-versa (ve Seção 6.1), então as regras de transformações podem ser armazenadas como sendo produções. Assim **prod1** indica a produção da linguagem 1 e **prod2** indica a produção da linguagem 2.

```
typedef struct tlistant {
        char *conteudo;
        TProduções *lpinicio, *lpfim;
        struct tlistant *prox, *ant;
} TListaNT;
```

A estrutura **TListaNT** é utilizada para armazenar todos os não-terminais da gramática. O campo **conteudo** armazena o símbolo não-terminal. Os campos **lpinicio** e **lpfim** indicam respectivamente a primeira e a última produção do não-terminal.

Os módulos **Arvore** e **Transforma** que são utilizados no momento da transformação de uma linguagem para outra trabalham sobre árvores de derivação armazenadas em árvores binárias que representam árvores n-árias. Esta representação é feita da seguinte maneira: "os filhos de um nó em uma árvore n-ária são armazenados como filhos da esquerda deste nó na árvore binária; os irmãos de um nó na árvore n-ária são armazenados como filhos da direita deste nó na árvore binária" [HOR86].

```
typedef struct tarvore {
     char *conteudo;
     short tipo;
     short regra;
     struct tarvore *fesq;
     struct tarvore *fdir;
} TArvore;
```

Na estrutura **TArvore** é armazenada a árvore de derivação da linguagem de entrada e da linguagem de saída. No campo **tipo** está indicado o que o nó representa, um terminal, um não-terminal, uma ação semântica TOP-DOWN ou uma ação semântica BOTTOM-UP. O campo ***fesq** indica o primeiro filho deste nó. O campo ***fdir** indica o primeiro irmão deste nó. O campo **regra** especifica através de qual regra que o não-terminal foi reduzido.

As estruturas de dados utilizadas pelo módulo **Transforma** estão apresentadas no Anexo C.

6.5.2 Algoritmos

6.5.2.1 Passos de execução do UCC

A seguir é apresentada a sequência de operações que são realizadas pelo usuário ou pelo UCC até o compilador da linguagem 1 para a linguagem 2 estar pronto.

Passo 1:

O usuário descreve a gramática da linguagem de entrada com as trasformações para a linguagem 2.

Passo 2:

O usuário submete o arquivo contendo a gramática descrita no passo anterior ao UCC. O usuário informa

também se o compilador a ser gerado é da linguagem 1 para a linguagem 2, ou vice-versa.

Passo 3:

O UCC processa o arquivo contendo a gramática descrita no passo 1 e gera os arquivos:

GRAM.YACC: contém a descrição da gramática para o YACC

ESTRUT.H: contém as estruturas para o módulo **Transforma** (ver Anexo C).

ERRO.UCC: contém eventuais erros encontrados durante a análise da gramática descrita no passo 1.

Passo 4:

O UCC submete o arquivo GRAM.YACC para o YACC.

Passo 5:

O YACC gera o arquivo Y.TAB.C que contém o *parser* para a gramática de entrada (conforme selecionado pelo usuário no passo 2).

Passo 6:

O usuário submete os arquivos:

TRANSFORMA.C
ARVORE.C
Y.TAB.C
RotinasDoUsuario.C
Lexico.C

a um compilador C, obtendo assim um compilador da linguagem 1 para a linguagem 2, ou vice-versa.

Passo 7.

O usuário executa o compilador gerado no passo 7 sobre uma sentença de entrada. O compilador gera um arquivo contendo a sentença de entrada transformada.

6.5.2.2 Avaliação de atributos

Na Seção 6.4 é descrito como o UCC pode trabalhar com atributos herdados e sintetizados. A avaliação destes atributos está relacionado com a ordem na qual as ações semânticas BOTTOM-UP e TOP-DOWN são executadas.

As ações semânticas BOTTOM-UP são executadas durante o processo de reconhecimento da entrada. Sempre que ocorrer uma redução [AHO86] é verificado se existe uma ação semântica associada a produção que está sendo reduzida, se existir, então tal ação é executada.

Depois de reconhecida a sentença de entrada e gerada a árvore de derivação da entrada as ações semânticas TOP-DOWN são executadas. Vale salientar que estas ações são executadas após todas as ações semânticas BOTTOM-UP, relacionadas com alguma produção reduzida, terem sido executadas.

Conforme apresentado na Seção 6.4 pode-se ter atributos herdados e sintetizados usando qualquer um dos tipos de ação semântica. A herança de atributos pode ser de 3 formas:

A. herdar um atributo do símbolo da esquerda da produção (herdar do pai na árvore de derivação [AHO86]). Nesta forma pode-se utilizar ações semânticas BOTTOM-UP e TOP-DOWN (ver Gramática 6.2 e 6.3 respectivamente);

B. se A e B são símbolos da direita de uma produção na forma A B, ou seja A está a esquerda de B (A é irmão da esquerda de B na árvore de derivação), então B pode herdar o atributo de A, pois o atributo de A já está calculado.

```
PROD : A [ $2.atr = $1.atr; ] B -> $1.atr $2.atr;

A    : 'a' [ $$.atr = 'a'; ]
     | 'A' [ $$.atr = 'A'; ];

B    : 'b' [ if ($$.atr == 'a') $$.atr = 'B';
             else $$.atr = 'b'; ]
     | 'B' [  if ($$.atr == 'A') $$.atr = 'b';
             else $$.atr = 'B'; ];
```

Gramática 6.4 Combinação de um A e um B (1)

Abaixo são apresentadas as sentenças aceitas para a gramática acima e a transformação realizada em cada uma das sentenças.

Sentença de entrada	Sentença transformada
ab	aB
aB	aB
Ab	Ab
AB	Ab

Na gramática acima o valor do atributo de B depende do valor do atributo de A. Se A tem como atributo 'a' (minúsculo) então B tem como atributo 'B' (maiúsculo). Se A tem como atributo 'A' (maiúsculo) então B tem como atributo 'b' (minúsculo).

C. se A e B são símbolos da direita de uma produção na forma A B, ou seja B está a direita de A (A é irmão da direita de B na árvore de derivação), então A pode herdar o atributo de B, pois o atributo de B pode ja ter sido calculado por uma ação semântica BOTTOM-UP e o atributo de A será calculado por uma ação semântica TOP-DOWN.

```
PROD : [ $1.atr = $2.atr; ] A B -> $1.atr $2.atr;

A    : 'a' [ if ($$.atr == 'B') $$.atr = 'A';
             else $$.atr = 'a'; ]
     | 'A' [  if ($$.atr == 'b') $$.atr = 'a';
             else $$.atr = 'A'; ];

B    : 'b' { $$.atr = 'b'; }
     | 'B' { $$.atr = 'B'; };
```

Gramática 6.5 Combinação de um A e um B (2)

Abaixo são apresentadas as sentenças aceitas para a gramática acima e a transformação realizada em cada uma das sentenças.

Sentença de entrada	Sentença transformada
ab	ab
aB	AB
Ab	ab
AB	AB

Na gramática acima o valor do atributo de A depende do valor do atributo de B. Se B tem como atributo 'b' (minúsculo) então A tem como atributo 'a' (minúsculo). Se B tem como atributo 'B' (maiúsculo) então A tem como atributo 'A' (maiúsculo).

6.5.2.3 Algoritmos para transformação

Algoritmo 1 - Transforma produção da árvore de derivação

```
Entrada:  árvore de derivação
          regra de transformação
Saída:    árvore de derivação transformada
0. Início
1. cria um nó S para a árvore transformada
2. para cada item na regra de transformação associado ao
   Não-Terminal da árvore de entrada faça
   3. Se o item é um terminal
      então faça
          4. cria um novo nó R
          5. insere este nó como filho da esquerda de S
      senão
          6. se item é um Não-Terminal
             então
               7. se ele se encontra na árvore de entrada
                    como filho da raiz
                  então
                    8. copia ele como filho da esquerda
                       de S
                  senão ERRO

9. retorna S
10. Fim
```

Algoritmo 2 - Transforma toda árvore de derivação

Entrada: árvore de derivação

Saída: árvore de derivação transformada

0. Início
1. se árvore de derivação da entrada não é vazia
   então faça
   2. aplica sobre raiz da árvore de derivação da
      entrada o Algoritmo 1
   3. raiz da entrada recebe o valor retornado pelo
      Algoritmo 1
   4. para cada filho da raiz da árvore de derivaçao
      entrada faça
      5. aplicar o Algoritmo 1 sobre o filho
6. retorna S
7. Fim

7. VALIDAÇÃO

A ferramenta apresentada teve como objetivo principal a apresentação da viabilidade da utilização de Gramáticas Transformacionais com Atributos. Com o UCC mostrou-se que é possível construir uma ferramenta que engloba os conceitos de Gramáticas Transformacionais e de Gramáticas de Atributos.

Além de mostrar a viabilidade da utilização destas duas gramáticas, problemas apresentados por outras ferramentas, principalmente pelas estudadas no Capítulo 5, foram resolvidos, e com baixos custos, tanto de implementação como de recursos de máquina.

A dificuldade que ferramentas do estilo do YACC e do SINLEX possuem em realizar transformações, conforme visto no Capítulo 5, ou seja a necessidade do usuário programar as regras transformacionais com uma linguagem de programação, foi eliminada. Na gramática abaixo, com o formato do YACC, mostra-se a dificuldade de transformar uma expressão infixada, na sua correspondente notação polonesa.

```
EXPR : EXPR "+" EXPR
          { strcpy($$.str,$1.str);
            strcat($$.str,$3.str);
            strcat($$.str,"+");
          }
     | EXPR "*" EXPR
          { strcpy($$.str,$1.str);
            strcat($$.str,$3.str);
            strcat($$.str,"*");
          }
     | id { strcpy($$.str,id); };
```

Neste exemplo o usuário deve fazer as concatenações das expressões através da operação *strcat*, e de cópias do

atributo das expressões para o atributo do símbolo E do lado esquerdo da produção.

Com a utilização do UCC, a gramática anterior é simplificada para:

```
E : E "+" E -> E E "+"
  | E "*" E -> E E "*"
  | id -> id;
```

Além de ser de entendimento mais claro, o usuário não necessita conhecer a linguagem de programação C para realizar as transformações.

Outra grande dificuldade encontrada nas ferramentas estudadas, SINLEX e GG, é que estas utilizam gramáticas ELL(1) para descrever a linguagem de entrada. Esta dificuldade está associada ao fato das gramáticas ELL(1) não permitirem a utilização de recursividade à esquerda nas produções, o que muitas vezes reduz drasticamente a legibilidade e a naturalidade da descrição da linguagem de entrada. Em [ZOR91] é apresentado uma ferramenta de conversão de Pascal para C utilizando a ferramenta GG. Durante a descrição da gramática do Pascal, muitas das estruturas perderam sua clareza por terem sido fatoradas.

Se depender somente da recursividade à esquerda, a clareza da gramática de entrada não é perdida, uma vez que, no sistema UCC as gramáticas suportadas são as LALR(1), as quais permitem tal recursividade.

O sistema GG é uma ferramenta transformacional para geração de código objeto. Neste tipo de tradutor, que gera código objeto, é muito importante a criação de *Tabelas de Símbolos* durante a análise da sentença de entrada. Este fato não foi levado em conta quando da implementação da referida ferramenta. Existe a

possibilidade da criação destas tabelas, mas esta tarefa só pode ser realizada através das rotinas semânticas fornecidas pela ferramenta. Além de utilizar rotinas semânticas, o usuário (projetista) deve além de conhecer as estruturas de dados (árvore de derivação) nas quais os *tokens* da entrada estão armazenados, percorrer estas estruturas e só então inserir um determinado símbolo nas referidas tabelas.

O problema existente no GG para inserção de símbolos em tabelas, não existe no UCC, uma vez que ações semânticas são permitidas durante o processo de análise da sentença de entrada (ações BOTTOM-UP), e ainda após toda a árvore ter sido construída (ações TOP-DOWN).

Outro problema encontrado no sistema GG, é o desempenho para fazer a conversão entre duas linguagens. O sistema GG foi implementado sobre o ambiente operacional DOS dos microcomputadores IBM-PC Compatíveis, sendo portanto limitado e lento. No UCC este problema foi resolvido, no momento que a opção pelo ambiente operacional visava ser um ambiente que fornecesse recursos de memória e velociade compatíveis com suas necessidades. O UCC foi implementado sobre o sistema operacional UNIX das estações de trabalho Sparc SUN (conforme descrito na Seção 6.5).

8. SUGESTÕES

O presente trabalho mostrou a definição de uma GTA, e uma ferramenta foi construída para validar a proposta.

A partir desta ferramenta outros estudos podem ser elaborados, ou outras ferramentas podem ser construídas. Entre os estudos que podem ser realizados a partir da ferramenta construída, dois são os mais destacados:

-estudar uma linguagem canônica o mais abrangente possível para as linguagens de programação procedurais como o Pascal, C, Ada, ... Deste modo pode-se construir tradutores para quaisquer linguagens. Assim para 4 linguagens, tem-se no máximo 8 tradutores. Produz-se um tradutor de cada linguagem para a linguagem canônica, e da linguagem canônica para cada linguagem. Se tivesse que ser construído um tradutor de cada uma das linguagens para outra, então ter-se-ia 12 tradutores [TAN83].

-estudar a viabilidade da utilização das GTAs na transformação entre linguagens naturais. É comum a necessidade de converter textos escritos em uma linguagem natural para outra, sendo esta tarefa muito cansativa. Se este processo fosse automatizado então o tempo ganho seria extremamente elevado.

Estas duas idéias são as mais emergentes que o autor deste trabalho vislumbra. Inúmeras outras atividades podem ainda ser desenvolvidas sobre o assunto apresentado.

9. CONCLUSÃO

Este trabalho apresentou as Gramáticas Transformacionais e as Gramáticas com Atributos, e a partir destas duas foi criado um protótipo de ferramenta que trabalha sobre gramáticas derivadas destas duas, ou seja, as Gramáticas Transformacionais com Atributos (GTAs).

Com a ferramenta desenvolvida mostrou-se que é possível, utilizando as GTAs, fazer transformações entre linguagens de uma maneira simples e eficiente, podendo um usuário resolver qualquer problema que pode ser resolvido por uma Máquina de Touring [YEL88].

O trabalho surgiu a partir de uma Dissertação de Mestrado, e de diversas Teses de Doutorado, desenvolvidas no final da década de 80 e inicio de 90, o que mostra que este tema está no *estado da arte* e portanto muitos estudos ainda estão sendo realizados em paralelo com este.

A quantidade de áreas que podem utilizar o método de transformação apresentado é muito grande, incluíndo Compiladores, Microeletrônica, Linguagem Natural, e qualquer outra área que necessite da transformação de uma determinada entrada em uma sentença de saída.

ANEXO A - GRAMÁTICA TRANSFORMACIONAL DO PASCAL PARA C

Abaixo encontra-se um sub-conjunto da linguagem Pascal. Este sub-conjunto é descrito em [AHO86]. Já estão incluídas na gramática, as regras transformacionais (em negrito). Após a descrição da gramática e das regras de transformações é apresentado um pequeno exemplo de conversão.

```
PROGRAM :   "program" ident "(" IDENT_LIST ")" ";"
            DECLARATIONS
            SUBPROGRAM_DECLS
            COMPOUND_STAT "." ->
                  DECLARATIONS
                  SUBPROGRAM_DECLS
                  ident "()"
                  COMPOUND_STAT ;

IDENT_LIST  : ident [ if ($$.atr == -1) /** ident simples **/
                        $1.atr = "";
                      else {
                              strcpy($1.atr,"[");
                              strcat($1.atr,convnumstr($$.atr));
                              strcat($1.atr,"]");
                            }
                        ] -> ident $1.atr
```

```
            | IDENT_LIST "," ident
                 [ if ($$.atr == -1) /** ident simples **/
                       $1.atr = "";
                   else {
                          strcpy($1.atr,"[");
                          strcat($1.atr,convnumstr($$.atr));
                          strcat($1.atr,"]");
                        }
                 ] -> IDENT_LIST "," ident $1.atr;

DECLARATIONS: DECLARATIONS "var" [ $3.atr = $4.atr ] IDENT_LIST ":"
TYPE
                 ";" -> DECLARATIONS TYPE IDENT_LIST ";"
            | -> ;

TYPE   : STANDARD_TYPE { $$.atr = ""; } -> STANDARD_TYPE
       | "array" "[" num ".." num "]" { $$.atr = $5.val - $3.val + 1;}
            "of" STANDARD_TYPE -> STANDARD_TYPE ;

STANDARD_TYPE      : "integer" -> "int"
                   | "real"          -> "float";

SUBPROGRAM_DECLS : SUBPROGRAM_DECLS SUBPROGRAM_DECL ";" ->
                    SUBPROGRAM_DECLS SUBPROGRAM_DECL ";"
                   | -> ;

SUBPROGRAM_DECL : SUBPROGRAM_HEAD DECLARATIONS COMPOUND_STAT
              ->  SUBPROGRAM_HEAD DECLARATIONS COMPOUND_STAT ;

SUBPROGRAM_HEAD   :"function" ident ARGUMENTS ":" STANDARD_TYPE
                  ";" -> STANDARD_TYPE ident ARGUMENTS ";"
                   | "procedure" ident ARGUMENTS ";" ->
                              "void" ident ARGUMENTS ";" ;

ARGUMENTS    : "(" PARAM_LIST ")"     -> "(" PARAM_LIST ")"
             |                        -> ;
```

```
PARAM_LIST   : IDENT_LIST ":" TYPE   -> TYPE IDENT_LIST
             | PARAM_LIST ";" IDENT_LIST ":" TYPE ->
                          PARAM_LIST ";" TYPE IDENTI_LIST;

COMPOUND_STAT: "begin" OPT_STATS "end" -> "{" OPT_STATS "}" ;

OPT_STATS    : STAT_LIST -> STAT_LIST
             |           -> ;

STAT_LIST    : STAT                    -> STAT
             | STAT_LIST ";" STAT      -> STAT_LIST ";" STAT ;

STAT  : VARIABLE ":=" EXPR             -> VARIABLE "=" EXPR
      | PROCEDURE_STAT                 -> PROCEDURE_STAT
      | COMPOUND_STAT                  -> COMPOUND_STAT
      | "if" EXPR "then" STAT "else" STAT       ->
                              "if" "(" EXPR ")" STAT "else" STAT
      | "while" EXPR "do" STAT ->"while" "(" EXPR ")" STAT;

VARIABLE     : ident                   -> ident
             | ident "[" EXPR "]"      -> ident "[" expr "]";

PROCEDURE_STAT    : ident          -> ident
      | ident "(" IDENT_LIST ")" -> ident "(" IDENT_LIST ")"

EXPR_LIST    : EXPR                    -> EXPR
             | EXPR_LIST "," EXPR      -> EXPR_LIST "," EXPR ;

EXPR  : SIMPLE_EXPR                     -> SIMPLE_EXPR
      | SIMPLE_EXP RELOP SIMPLE_EXPR -> SIMPLE_EXPR RELOP
                                                SIMPLE_EXP;
```

```
SIMPLE_EXPR : TERM                          -> TERM
            | "+" TERM                      -> "+" TERM
            | "-" TERM                      -> "-" TERM
            | SIMPLE_EXPR ADDOP TERM        -> SIMPLE_EXPR ADDOP TERM;

TERM  : FACTOR                  -> FACTOR
      | TERM MULOP FACTOR       -> TERM MULOP FACTOR ;

FACTOR      : ident                     -> ident
            | ident "(" EXPR_LIST ")"-> ident "(" EXPR_LIST ")"
            | num                       -> num
            | "(" EXPR ")"              -> "(" EXPR ")"
            | "not" FACTOR ;            -> "!" FACTOR ;


RELOP : ">"  -> ">"
      | "<"  -> "<"
      | "<=" -> "<="
      | ">=" -> ">="
      | "="  -> "=="
      | "<>" -> "!=" ;

ADDOP : "+"  -> "+"
      | "-"  -> "-"
      | "or" -> "||";

MULOP : "*"   -> "*"
      | "/"   -> "/"
      | "and" -> "&&" ;
```

EXEMPLO 1:

Pascal:

```
Program teste;

Var a : integer;
     b,c,d : array[1..10] of char;
Begin
  a := 5;
End.
```

C

```
int a;
char b[10],c[10],d[10];

teste()
{
  a = 5;
}
```

ANEXO B - GRAMÁTICA DE ENTRADA DO UCC

A gramática de entrada do UCC descrita abaixo é entrada para a ferramenta YACC.

```
Inicio : NaoTerminal  ":" ListaSimbolos "->"
          ListaSimbolos Inicio_ ";" Inicio
          | ;
Inicio_ :"|" ListaSimbolos "->" ListaSimbolos Inicio_
          | ;

ListaSimbolos : NaoTerminal ListaSimbolos
               | Terminal ListaSimbolos
               | AcaoTopDown ListaSimbolos
               | AcaoBottomUp ListaSimbolos
               | ;
```

ANEXO C - EXEMPLO DE DESCRIÇÃO PARA O UCC COM OS ARQUIVOS GERADOS PELO UCC

Gramática de Entrada para o UCC

```
%%
line   :  expr -> expr;

expr   :  expr  "+" term   -> expr term "+"
          | term           -> term;

term   :  term "*"factor   -> term factor "*"
          | factor ;

factor : '(' expr ')'      -> expr
          | DIGIT          -> DIGIT;
```

Arquivo com as regras transformacionais em Estruturas de Dados para os módulos "Arvore" e "Transforma"

```
/*****************************************************************************
ESTRUT.H

        Estruturas que armazena os nao terminais e as regras de transformacoes.

        (c) UCC - Um Compilador de Compiladores
        Autor: Avelino F. Zorzo
        Data  : 16.12.92

*****************************************************************************/

#define MAXPROD 3

typedef struct termnterm {
        short nterm;     /** 0:terminal; outro numero:nao-terminal **/
        char *term;      /** se é terminal, ele é armazenado aqui **/
             }  TermNTerm;


typedef struct ttransf  {
        short quantia;                  /** Numero de itens na transformacao **/
        TermNTerm trans[MAXPROD];   /** Transformacoes **/
             }  TTransformacao;
```

```
TTransformacao yytransforma[] =
{
    { 0, { {0,""}, {0,""}, {0,""} } },    /** Nenhuma regra **/
    { 1, { {1,""}, {0,""}, {0,""} } },    /** line : expr -> expr **/
    { 3, { {1,""}, {3,""}, {0,"+"} } },   /** expr : expr  "+" term  -> expr term "+"  **/
    { 1, { {1,""}, {0,""}, {0,""} } },    /** expr : term -> term **/
    { 3, { {1,""}, {3,""}, {0,"*"} } },   /** term : term "*"factor  -> term factor "*" **/
    { 1, { {1,""}, {0,""}, {0,""} } },    /** expr : factor -> factor **/
    { 1, { {2,""}, {0,""}, {0,""} } },    /** factor : '(' expr ') -> expr **/
    { 1, { {1,""}, {0,""}, {0,""} } }     /** factor : DIGIT -> DIGIT **/
};


char *yynterminal[] =
{
        "-no no-terminal",
        "LINE",
        "EXPR",
        "TERM",
        "FACTOR"
};


/***** Fim do Arquivo ESTRUT.H  ******/
```

ANEXO D - PARSER DO YACC COM ALTERAÇÕES DO UCC

```
extern char *malloc(), *realloc();

#include <ctype.h>
typedef union { char v[80]; } YYSTYPE;

#define YYDEBUG 1

# define DIGIT 257
# define LF 258
# define ADD 259
# define MUL 260
#define yyclearin yychar = -1
#define yyerrok yyerrflag = 0
extern int yychar;
extern int yyerrflag;
#ifndef YYMAXDEPTH
#define YYMAXDEPTH 150
#endif
YYSTYPE yylval, yyval;
# define YYERRCODE 256


int yyexca[] ={
-1, 1,
        0, -1,
        -2, 0,
        };
# define YYNPROD 8
# define YYLAST 221
int yyact[]={

    8,  12,   7,   5,   4,   2,   3,   1,   0,   0,
    0,   9,   0,  11,  10,   0,   0,   0,   0,   0,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
```

```
    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
    0,    0,    0,    0,    0,    0,    0,    0,    0,    7,
    6 };
int yypact[]={

  -37, -1000,  -257,  -260, -1000,   -37, -1000,   -37,   -37,   -40,
 -260, -1000, -1000 };
int yypgo[]={

    0,    7,    5,    6,    4 };
int yyr1[]={

    0,    1,    2,    2,    3,    3,    4,    4 };
int yyr2[]={

    0,    3,    7,    3,    7,    3,    7,    3 };
int yychk[]={

 -1000,   -1,   -2,   -3,   -4,   40,  257,  259,  260,   -2,
   -3,   -4,   41 };
int yydef[]={

    0,   -2,    1,    3,    5,    0,    7,    0,    0,    0,
    2,    4,    6 };
typedef struct { char *t_name; int t_val; } yytoktype;
#ifndef YYDEBUG
#       define YYDEBUG         0       /* don't allow debugging */
#endif

#if YYDEBUG

yytoktype yytoks[] =
{
        "DIGIT",        257,
        "LF",   258,
        "ADD",  259,
        "MUL",  260,
        "-unknown-",    -1      /* ends search */
};

char * yyreds[] =
{
        "-no such reduction-",
        "line : expr",
        "expr : expr ADD term",
        "expr : term",
        "term : term MUL factor",
        "term : factor",
        "factor : '(' expr ')'",
        "factor : DIGIT",
};
#endif /* YYDEBUG */
#line 1 "/usr/lib/yaccpar"
/*      @(#)yaccpar 1.10 89/04/04 SMI; from S5R3 1.10           */
```

```
/*
** Skeleton parser driver for yacc output
*/

/*
** yacc user known macros and defines
*/
#define YYERROR         goto yyerrlab
#define YYACCEPT    { free(yys); free(yyv); return(0); }
#define YYABORT         { free(yys); free(yyv); return(1); }
#define YYBACKUP( newtoken, newvalue )\
{\
        if ( yychar >= 0 || ( yyr2[ yytmp ] >> 1 ) != 1 )\
        {\
                yyerror( "syntax error - cannot backup" );\
                goto yyerrlab;\
        }\
        yychar = newtoken;\
        yystate = *yyps;\
        yylval = newvalue;\
        goto yynewstate;\
}
#define YYRECOVERING()  (!!yyerrflag)
#ifndef YYDEBUG
#       define YYDEBUG      1       /* make debugging available */
#endif

/*
** user known globals
*/
int yydebug;                    /* set to 1 to get debugging */

/*
** driver internal defines
*/
#define YYFLAG          (-1000)

/*
** static variables used by the parser
*/
static YYSTYPE *yyv;                    /* value stack */
static int *yys;                /* state stack */

static YYSTYPE *yypv;                   /* top of value stack */
static int *yyps;               /* top of state stack */

static int yystate;                     /* current state */
static int yytmp;               /* extra var (lasts between blocks) */

int yynerrs;                    /* number of errors */

int yyerrflag;                  /* error recovery flag */
int yychar;                     /* current input token number */


/*
```

```
** yyparse - return 0 if worked, 1 if syntax error not recovered from
*/
int
yyparse()
{
        register YYSTYPE *yypvt;        /* top of value stack for $vars */
        unsigned yymaxdepth = YYMAXDEPTH;

        /*
        ** Initialize externals - yyparse may be called more than once
        */
        yyv = (YYSTYPE*)malloc(yymaxdepth*sizeof(YYSTYPE));
        yys = (int*)malloc(yymaxdepth*sizeof(int));
        if (!yyv || !yys)
        {
                yyerror( "out of memory" );
                return(1);
        }
        yypv = &yyv[-1];
        yyps = &yys[-1];
        yystate = 0;
        yytmp = 0;
        yynerrs = 0;
        yyerrflag = 0;
        yychar = -1;

        goto yystack;
        {
                register YYSTYPE *yy_pv;        /* top of value stack */
                register int *yy_ps;            /* top of state stack */
                register int yy_state;          /* current state */
                register int  yy_n;             /* internal state number info */

                /*
                ** get globals into registers.
                ** branch to here only if YYBACKUP was called.
                */
        yynewstate:
                yy_pv = yypv;
                yy_ps = yyps;
                yy_state = yystate;
                goto yy_newstate;

                /*
                ** get globals into registers.
                ** either we just started, or we just finished a reduction
                */
        yystack:
                yy_pv = yypv;
                yy_ps = yyps;
                yy_state = yystate;

                /*
                ** top of for (;;) loop while no reductions done
                */
        yy_stack:
```

```
                /*
                ** put a state and value onto the stacks
                */
#if YYDEBUG
                /*
                ** if debugging, look up token value in list of value vs.
                ** name pairs.  0 and negative (-1) are special values.
                ** Note: linear search is used since time is not a real
                ** consideration while debugging.
                */
                if ( yydebug )
                {
                        register int yy_i;

                        (void)printf( "State %d, token ", yy_state );
                        if ( yychar == 0 )
                                (void)printf( "end-of-file\n" );
                        else if ( yychar < 0 )
                                (void)printf( "-none-\n" );
                        else
                        {
                                for ( yy_i = 0; yytoks[yy_i].t_val >= 0;
                                        yy_i++ )
                                {
                                        if ( yytoks[yy_i].t_val == yychar )
                                                break;
                                }
                                (void)printf( "%s\n", yytoks[yy_i].t_name );
                        }
                }
#endif /* YYDEBUG */
                if ( ++yy_ps >= &yys[ yymaxdepth ] )    /* room on stack? */
                {
                        /*
                        ** reallocate and recover.  Note that pointers
                        ** have to be reset, or bad things will happen
                        */
                        int yyps_index = (yy_ps - yys);
                        int yypv_index = (yy_pv - yyv);
                        int yypvt_index = (yypvt - yyv);
                        yymaxdepth += YYMAXDEPTH;
                        yyv = (YYSTYPE*)realloc((char*)yyv,
                                yymaxdepth * sizeof(YYSTYPE));
                        yys = (int*)realloc((char*)yys,
                                yymaxdepth * sizeof(int));
                        if (!yyv || !yys)
                        {
                                yyerror( "yacc stack overflow" );
                                return(1);
                        }
                        yy_ps = yys + yyps_index;
                        yy_pv = yyv + yypv_index;
                        yypvt = yyv + yypvt_index;
                }
                *yy_ps = yy_state;
                *++yy_pv = yyval;
```

```
                /*
                ** we have a new state - find out what to do
                */
        yy_newstate:
                if ( ( yy_n = yypact[ yy_state ] ) <= YYFLAG )
                        goto yydefault;         /* simple state */
#if YYDEBUG
                /*
                ** if debugging, need to mark whether new token grabbed
                */
                yytmp = yychar < 0;
#endif
                if ( ( yychar < 0 ) && ( ( yychar = yylex() ) < 0 ) )
                        yychar = 0;             /* reached EOF */
#if YYDEBUG
                if ( yydebug && yytmp )
                {
                        register int yy_i;

                        (void)printf( "Received token " );
                        if ( yychar == 0 )
                                (void)printf( "end-of-file\n" );
                        else if ( yychar < 0 )
                                (void)printf( "-none-\n" );
                        else
                        {
                                for ( yy_i = 0; yytoks[yy_i].t_val >= 0;
                                        yy_i++ )
                                {
                                        if ( yytoks[yy_i].t_val == yychar )
                                                break;
                                }
                                (void)printf( "%s\n", yytoks[yy_i].t_name );
                        }
                }
#endif /* YYDEBUG */
                if ( ( ( yy_n += yychar ) < 0 ) || ( yy_n >= YYLAST ) )
                        goto yydefault;
                if ( yychk[ yy_n = yyact[ yy_n ] ] == yychar )       /*valid shift*/
                {
```

```
/**** Inserido pelo (c) UCC - Um Compilador de Compiladores ****/

        EmpilhaNodo(CriaNodoArvore(yylval,1));

/****-----------------------------------***/

                        yychar = -1;
                        yyval = yylval;
                        yy_state = yy_n;
                        if ( yyerrflag > 0 )
                                yyerrflag--;
                        goto yy_stack;
                }

        yydefault:
                if ( ( yy_n = yydef[ yy_state ] ) == -2 )
                {
#if YYDEBUG
                        yytmp = yychar < 0;
#endif
                        if ( ( yychar < 0 ) && ( ( yychar = yylex() ) < 0 ) )
                                yychar = 0;             /* reached EOF */
#if YYDEBUG
                        if ( yydebug && yytmp )
                        {
                                register int yy_i;

                                (void)printf( "Received token " );
                                if ( yychar == 0 )
                                        (void)printf( "end-of-file\n" );
                                else if ( yychar < 0 )
                                        (void)printf( "-none-\n" );
                                else
                                {
                                        for ( yy_i = 0;
                                                yytoks[yy_i].t_val >= 0;
                                                yy_i++ )
                                        {
                                                if ( yytoks[yy_i].t_val
                                                        == yychar )
                                                {
                                                        break;
                                                }
                                        }
                                        (void)printf( "%s\n", yytoks[yy_i].t_name );
                                }
                        }
#endif /* YYDEBUG */
                        /*
                        ** look through exception table
                        */
                        {
                                register int *yyxi = yyexca;

                                while ( ( *yyxi != -1 ) ||
```

```
                        ( yyxi[1] != yy_state ) )
                {
                        yyxi += 2;
                }
                while ( ( *(yyxi += 2) >= 0 ) &&
                        ( *yyxi != yychar ) )
                        ;
                if ( ( yy_n = yyxi[1] ) < 0 )
                        YYACCEPT;
        }
}

/*
** check for syntax error
*/
if ( yy_n == 0 )  /* have an error */
{
        /* no worry about speed here! */
        switch ( yyerrflag )
        {
        case 0:         /* new error */
                yyerror( "syntax error" );
                goto skip_init;
        yyerrlab:
                /*
                ** get globals into registers.
                ** we have a user generated syntax type error
                */
                yy_pv = yypv;
                yy_ps = yyps;
                yy_state = yystate;
                yynerrs++;
        skip_init:
        case 1:
        case 2:         /* incompletely recovered error */
                        /* try again... */
                yyerrflag = 3;
                /*
                ** find state where "error" is a legal
                ** shift action
                */
                while ( yy_ps >= yys )
                {
                        yy_n = yypact[ *yy_ps ] + YYERRCODE;
                        if ( yy_n >= 0 && yy_n < YYLAST &&
                                yychk[yyact[yy_n]] == YYERRCODE)
                {
                                /*
                                ** simulate shift of "error"
                                */
                                yy_state = yyact[ yy_n ];
                                goto yy_stack;
                        }
                        /*
                        ** current state has no shift on
                        ** "error", pop stack
```

```
                                        */
#if YYDEBUG
#       define _POP_ "Error recovery pops state %d, uncovers state %d\n"
                                        if ( yydebug )
                                                (void)printf( _POP_, *yy_ps,
                                                        yy_ps[-1] );
#       undef _POP_
#endif
                                        yy_ps--;
                                        yy_pv--;
                                }
                                /*
                                ** there is no state on stack with "error" as
                                ** a valid shift.  give up.
                                */
                                YYABORT;
                        case 3:         /* no shift yet; eat a token */
#if YYDEBUG
                                /*
                                ** if debugging, look up token in list of
                                ** pairs.  0 and negative shouldn't occur,
                                ** but since timing doesn't matter when
                                ** debugging, it doesn't hurt to leave the
                                ** tests here.
                                */
                                if ( yydebug )
                                {
                                        register int yy_i;

                                        (void)printf( "Error recovery discards " );
                                        if ( yychar == 0 )
                                                (void)printf( "token end-of-file\n" );
                                        else if ( yychar < 0 )
                                                (void)printf( "token -none-\n" );
                                        else
                                        {
                                                for ( yy_i = 0;
                                                        yytoks[yy_i].t_val >= 0;
                                                        yy_i++ )
                                                {
                                                        if ( yytoks[yy_i].t_val
                                                                == yychar )
                                                        {
                                                                break;
                                                        }
                                                }
                                                (void)printf( "token %s\n",
                                                        yytoks[yy_i].t_name );
                                        }
                                }
#endif /* YYDEBUG */
                                if ( yychar == 0 )          /* reached EOF. quit */
                                        YYABORT;
                                yychar = -1;
                                goto yy_newstate;
                        }
```

```
                }/* end if ( yy_n == 0 ) */
                /*
                ** reduction by production yy_n
                ** put stack tops, etc. so things right after switch
                */
#if YYDEBUG
                /*
                ** if debugging, print the string that is the user's
                ** specification of the reduction which is just about
                ** to be done.
                */
                if ( yydebug )
                        (void)printf( "Reduce by (%d) \"%s\"\n",
                                yy_n, yyreds[ yy_n ] );
#endif
                yytmp = yy_n;                   /* value to switch over */
                yypvt = yy_pv;                  /* $vars top of value stack */
                /*
                ** Look in goto table for next state
                ** Sorry about using yy_state here as temporary
                ** register variable, but why not, if it works...
                ** If yyr2[ yy_n ] doesn't have the low order bit
                ** set, then there is no action to be done for
                ** this reduction.  So, no saving & unsaving of
                ** registers done.  The only difference between the
                ** code just after the if and the body of the if is
                ** the goto yy_stack in the body.  This way the test
                ** can be made before the choice of what to do is needed.
                */
                {
                        /* length of production doubled with extra bit */
                        register int yy_len = yyr2[ yy_n ];


/***             Inserido pelo (c) UCC - Um Compilador de Compiladores          ***/

        EmpilhaNodo(CriaSubArvore(yynterminal[ yyr1[yy_n] ] , (yy_len >> 1)),yy_n ) ;


/***-------------------------------------------****/

                        if ( !( yy_len & 01 ) )
                        {
                                yy_len >>= 1;
                                yyval = ( yy_pv -= yy_len )[1];    /* $$ = $1 */
                                yy_state = yypgo[ yy_n = yyr1[ yy_n ] ] +
                                        *( yy_ps -= yy_len ) + 1;
                                if ( yy_state >= YYLAST ||
                                        yychk[ yy_state =
                                        yyact[ yy_state ] ] != -yy_n )
                                {
                                        yy_state = yyact[ yypgo[ yy_n ] ];
                                }
                                goto yy_stack;
                        }
```

```
                        yy_len >>= 1;
                        yyval = ( yy_pv -= yy_len )[1];    /* $$ = $1 */
                        yy_state = yypgo[ yy_n = yyr1[ yy_n ] ] +
                                *( yy_ps -= yy_len ) + 1;
                        if ( yy_state >= YYLAST ||
                                yychk[ yy_state = yyact[ yy_state ] ] != -yy_n )
                        {
                                yy_state = yyact[ yypgo[ yy_n ] ];
                        }
                }
                                        /* save until reenter driver code */
                yystate = yy_state;
                yyps = yy_ps;
                yypv = yy_pv;
        }
        /*
        ** code supplied by user is placed in this switch
        */
        switch( yytmp )
        {

case 1:
# line 15 "e2.yacc"
{ printf("%s\n",yypvt[-0].v);} break;
case 2:
# line 17 "e2.yacc"
{  strcpy(yyval.v, yypvt[-2].v);
                    strcat(yyval.v,  yypvt[-0].v);
                    strcat(yyval.v,yypvt[-1].v);  } break;
case 3:
# line 20 "e2.yacc"
{ strcpy(yyval.v,yypvt[-0].v);   } break;
case 4:
# line 23 "e2.yacc"
{ strcpy(yyval.v,yypvt[-2].v);
                     strcat(yyval.v,yypvt[-0].v);
                      strcat(yyval.v,yypvt[-1].v); } break;
case 5:
# line 26 "e2.yacc"
{ strcpy(yyval.v,yypvt[-0].v); } break;
case 6:
# line 29 "e2.yacc"
{   strcpy(yyval.v,yypvt[-1].v); } break;
case 7:
# line 30 "e2.yacc"
{  strcpy(yyval.v,yypvt[-0].v); } break;
        }
        goto yystack;               /* reset registers in driver code */
}
```

BIBLIOGRAFIA

[AHO73] AHO, Alfred V.; ULLMAN, Jeffrey D. The Theory of parsing, translation, and compiling. Englewood Cliffs: Prentice-Hall, 1973. 2v. v.1:Parsing.

[AHO73a] AHO, Alfred V.; ULLMAN, Jeffrey D. The Theory of parsing, translation, and compiling. Englewood Cliffs: Prentice-Hall, 1973. 2v. v.2:Compiling.

[AHO77] AHO, Alfred V.; ULLMAN, Jeffrey D.. Principles of compiler design, New Jersey: Addison-Wesley, 1977. 604p.

[AHO86] AHO, Alfred V.; ULLMANN, Jeffrey. Compilers: principles, techniques and tools. Reading: Addison-Wesley, 1977. 796p.

[AZE87] AZEREDO, Paulo A.. Geração automática de código objeto. Porto Alegre: CPGGC da UFRGS, 1987. 32p. (Relatório Interno).

[BAC64] BACH, Emmon. An introduction to transformational grammars. New York: Rinehart and Winston, 1964. 205p.

[BAU74] BAUER, F.L, et al. Compiler construction: an advanced course. Berlin: Springer-Verlag, 1974. 621p. (Lecture Notes in Computer Science, 21).

[CAS90] CASTRO, Paulo R. P.. Um protótipo de ferramenta transformacional, Porto Alegre: CPGCC da UFRGS, 1990. 55p. (Trabalho Individual, 175).

[CLA76] CLARK, Keith; COWELL Don. Programs, machines and computation: an introduction to the theory of computing. Londres: McGraw-Hill, 1976. 176p.

[DER74] DEREMER, F.L.. Transformational Grammars. In: Compiler construction: an advanced course. Berlin: Springer-Verlag, 1974. 621p. (Lecture Notes in Computer Science, 21).

[DER88] DERANSART, Pierre; JOURDAN, Martin; LORHO, Bernard. Attribute grammars: definitions, systems and bibliography. Berlin: Springer-Verlag, 1988. 232p. (Lecture Notes in Computer Science, 323).

[FAR87] FARROW, Rodney; YELLIN, Daniel. Generalized inversion of translation specifications. Yorktown Heights: IBM, 1987. 45p.

[GAN85] GANAPATHI, Mahadevan; FISCHER, Charles N.. Affix grammar driven code generation. ACM Transactions on programming languages and systems, v.7, n.4, p.560-599, Oct. 1985.

[HOP79] HOPCROFT, John E.; ULLMANn, Jeffrey D., Introduction to automata theory, languages, and computation. Reading: Addison-Wesley, 1979. 416p.

[HOR87] HORSPOOL, R.N.; LEVY, Michael R.. MkScan - An interactive scanner generator. Software Practice and Experience, v.17, n.6, p.369-378, June 1987.

[JOH75] JOHNSON, S.C.. Yacc - Yet another compiler compiler. New Jersey: AT&T Bell Laboratories, 1975. 73p. (Computing Science Technical Report, 32).

[KIN83] KING, M.. Transformational Parsing. In: Lugano tutorial on parsing natural languages. 2., July 6-11, 1981, Lugano. London: Academic Press, 1983. 308p.

[KOU66] KOUTSOUDAS, Andreas. Writing Transformational Grammars: an introduction. New York: McGraw-Hill, 1966. 368p.

[QUA89] QUADROS, Felipe S.. Uma interface para gerador de código objeto. Porto Alegre: CPGCC da UFRGS, 1989. 34p. (Trabalho Individual).

[QUA91] QUADROS, Felipe S.. Um gerador automático de código objeto usando gramáticas transformacionais. Porto Alegre: CPGCC da UFRGS, 1991. 175p. (Dissertação de Mestrado).

[ROS89] ROSA, Fernando R.. SINLEX - Um ambiente de desenvolvimento de processadores de linguagens. Porto Alegre: CPGCC da UFRGS, 1989. 94p. (Projeto de Diplomação).

[SAL73] SALOMAA, A.. Formal Languages. ACM Monograph Series. New York: Academic Pres, 1973. 322p.

[SUN90] SUN MICROSYSTEMS Inc. Programmer's overview: utilities & libraries, 1990. 532p.

[TAN83] TANENBAUM, A.S. et al. A practical toolkit for making portable compilers. Communications of the ACM, v.26, n.9, p.654-660, Sept. 1983.

[TOF90] TOFTE, Mads. Compiler Generators. In: Monographs on Theoretical Computer Science. Berlin: Springer-Verlag, 1990. 146p.

[YEL88] YELLIN, Daniel. Attribute grammar inversion and source-to-source translation. Berlin: Springer-Verlag, 1988. 176p. (Lecture Notes in Computer Science, 302).

[ZOR91] ZORZO, Avelino F.. TP2C - Conversor de Turbo Pascal para C Usando Gramática Transformacionais. Porto Alegre: CPGCC da UFRGS, Abr. 1991. 100p. (Trabalho Individual, 222).

Informática
UFRGS

Gramáticas Transformacionais com Atributos.

Dissertação apresentada aos Senhores:

Profa. Dra. Ana Maria de Alencar Price

Prof. Dr. Paulo Alberto de Azeredo

Prof. Dr. Roberto Tom Price

Profa. Dra. Vera Lúcia Strube de Lima (PUC-RS)

Vista e permitida a impressão.
Porto Alegre, 19/10/94.

Prof. Dr. Paulo Alberto de Azeredo,
Orientador.

Prof. Dr. José Palazzo Moreira de Oliveira,
Coordenador do Curso de Pós-Graduação
em Ciência da Computação.