1.5.4.2. Estruturas de Controle: Nível de Unidades de Programação

As estruturas de controle no nível de unidades de programação são mecanismos de linguagens utilizados para especificar o fluxo de controle entre unidades de um programa. O mecanismo mais simples é o bloco, que cria um novo escopo de definição de objetos, sendo ativado e executado quando é encontrado durante o processo de execução seqüencial do programa. Mecanismos mais poderosos permitem ao programador transferir o controle de uma unidade para outra, através de uma chamada explícita a nova unidade.

Na maioria dos casos, a unidade chamada está subordinada à que a chamou, ou seja, após sua execução a unidade chamada retorna o controle a unidade que a chamou. Este é o caso de subprogramas, onde o retorno é feito através de uma operação do tipo *return*. No caso de linguagens que possuam facilidades de tratamento de exceções, a transferência de controle é implícita. A execução de uma unidade pode provocar uma exceção no processamento e implicitamente o controle é passado a um tratador de exceção associado àquele tipo de exceção (seção 1.6).

As unidades de um programa podem ser organizadas em um esquema simétrico, através de um conjunto de co-rotinas, onde as unidades passam explicitamente o controle de uma para outra. Finalmente, as unidades também podem ser organizadas como um conjunto de unidades concorrentes, ou processos. Neste caso, não existe a noção de passagem de controle de uma unidade para outra, ou seja, as unidades são consideradas autônomas, sendo executadas assincronamente (seção 4.2) [SIL 88]. Como tratamento de exceções e programação concorrente são discutidos em outras seções, agora serão descritos os mecanismos de subprogramas e blocos, e co-rotinas.

1) Subprogramas e Blocos

As linguagens de programação fornecem meios para agrupar instruções implementando uma abstração de ação em uma unidade de programa apropriada. O exemplo mais comum e útil é o subprograma, presente desde a formulação das primeiras linguagens de montagem. A abstração representada pelos subprogramas corresponde ao mapeamento de um conjunto de entradas num conjunto de saídas que pode ser descrito por uma especificação. Tal especificação define como as saídas se relacionam com as entradas, porém, não é necessário revelar a maneira como as saídas são processadas. Um subprograma é uma abstração porque permite a seus usuários concentrarem-se no que é feito e não no como é implementado. Essa classe cobre desde as sub-rotinas e funções do *Fortran*, até os procedimentos de *Ada* [GHE 87, SIL 88].

Então, um subprograma consiste num mecanismo de decomposição de programa que permite que os programas sejam divididos em várias unidades. Chamadas de subprogramas são estruturas de controle que governam o fluxo de controle entre estas unidades. As relações entre subprogramas definidas através de chamadas são assimétricas: a unidade que invoca outra unidade transfere o controle para a unidade chamada, a qual, após completar sua tarefa, devolve o controle ao ponto de chamada [GHE 97, SIL 88].

A maioria das LP fazem distinção entre dois tipos de rotinas (ou subprogramas): procedimentos e funções. Um procedimento não retorna um valor; ele é um comando abstrato que é chamado para fazer alguma troca de estado desejada. O estado pode mudar porque os valores de alguns parâmetros transmitidos ao procedimento sofrem modificações, pois algumas variáveis não locais são atualizadas pelo procedimento e algumas ações são desempenhadas no ambiente externo (exemplo: leitura ou escrita). Numa função, em contraposição, supõe-se que um valor, que depende do valor dos parâmetros, deve ser retornado. *Pascal*, assim como *Ada*, fornece tanto procedimentos como funções. Já em *C* todos os subprogramas são funções, isto é, retornam algum valor, a menos que o tipo retornado seja *void*, que identifica explicitamente que nenhum valor é retornado [GHE 97].

Um subprograma é caracterizado por quatro elementos: seu nome, uma lista de parâmetros, um corpo e um ambiente. Um subprograma é normalmente definido por uma palavra chave que informa ao compilador tratar-se de um subprograma, seguido do seu nome, da definição de uma lista de identificadores (“parâmetros formais”) e do seu corpo. Os identificadores contidos na lista não são variáveis, mas meramente nomes, que definem o papel que será exercido pelos “parâmetros reais” passados quando o subprograma for chamado. Normalmente, existe uma correspondência posicional (baseada na ordem em que aparecem na lista) entre os parâmetros reais e formais. Entretanto, algumas linguagens permitem, além do esquema posicional, associar os parâmetros reais aos formais através dos seus nomes.

A especificação dos parâmetros de um subprograma contém, normalmente, além do nome dos parâmetros formais, seu tipo e a maneira como eles serão utilizados em tempo de execução, que corresponde a passagem de parâmetros. Passagem de parâmetros é utilizada para comunicação de objetos entre unidades de programas. Distintamente de comunicação entre unidades via ambiente global, parâmetros permitem a transferência de objetos diferentes a cada chamada e constituem um meio mais adequado, considerando os aspectos de legibilidade dos programas e de manutenção. Duas classes de objetos podem ser repassados como parâmetros: dados e subprogramas [SIL 88].

Existem três convenções distintas para passagem de dados como parâmetros a um subprograma:

- Por referência: a unidade que chamou passa à unidade chamada o endereço do parâmetro real. Uma referência ao parâmetro, na unidade chamada, é tratada como uma referência à posição cujo endereço é passado. Portanto, uma variável passada por referência pode ser modificada diretamente pelo subprograma chamado.
- Por cópia: é feita uma cópia do valor do parâmetro real para o parâmetro formal. Assim, os parâmetros formais não compartilham memória com os parâmetros reais; em vez disso, eles agem como uma variável local. A passagem por cópia tem a vantagem de proteger os parâmetros reais de modificações inadvertidas (efeitos colaterais). Existem três maneiras de passar parâmetros por cópia: por valor, por resultado e por valor-resultado. (1) Na passagem por valor, os valores dos parâmetros reais são utilizados para inicializar os parâmetros formais correspondentes, que agem como variáveis locais na unidade chamada. Não é permitido retorno de informações à unidade que chamou através dos parâmetros. (2) Na passagem por resultado, os valores dos parâmetros formais são copiados para os parâmetros reais no término de execução da unidade chamada. Não permite o fluxo de informações da unidade que chamou para a chamada. (3) A passagem por valor-resultado compreende a fusão das duas anteriores, ou seja, os parâmetros formais são variáveis locais que são inicializadas com os valores dos parâmetros reais no momento da chamada (como na passagem por valor) e, no término da unidade chamada, os valores finais dos parâmetros formais são copiados para os parâmetros reais (como passagem de resultado).
- Por nome: como na passagem de referência, um parâmetro denota uma posição no ambiente da unidade que chamou, não sendo uma variável local; entretanto a amarração entre os parâmetros formais e reais não é feita na inicialização da unidade que chamou, mas toda vez que o parâmetro formal é usado. Como conseqüência, cada utilização de um parâmetro formal pode referenciar uma localização diferente. Com essas características pode-se observar que se os parâmetros reais forem variáveis escalares, a chamada por nome será equivalente à chamada por referência. Se o parâmetro real for uma expressão, a expressão será avaliada cada vez que o parâmetro formal for utilizado, podendo produzir valores diferentes a cada vez (basta que existam variáveis na expressão, cujos valores tenham sido modificados entre as duas avaliações).

A passagem de subprogramas como parâmetros é útil em muitas situações práticas. Por exemplo, se a linguagem não proporciona mecanismos explícitos para o tratamento de exceções, um subprograma tratador de exceções pode ser transmitido como parâmetro à unidade que pode acusar a exceção. Entretanto, embora útil, subprogramas como parâmetros podem facilmente levar à criação de programas obscuros, com pouca legibilidade.

Transmitir um subprograma como um parâmetro requer no mínimo a passagem de uma referência ao segmento de código do argumento. Entretanto, o subprograma passado como parâmetro pode, por sua vez, acessar variáveis não locais, sendo portanto necessário passar informações sobre o ambiente não local do argumento.

Em linguagens do gênero *Algol*, onde os registros de ativação são organizados em pilha, um parâmetro do tipo subprograma é representado pelo par (“pc”, “pa”), onde “pc” é um ponteiro para o segmento de código e “pa” é um ponteiro para o registro de ativação da unidade onde esse subprograma foi definido. Quando for feita uma chamada para o subprograma parâmetro, o seu registro de ativação é instalado no topo da pilha, o *link* estático assume o valor de “pa” e o controle de execução é passado ao código especificado por “pc”. O próximo exemplo em *Pascal* ilustra a passagem de um subprograma como parâmetro [GHE 87, SIL 88]:

```
procedure principal
    ...
    procedure a ...
        ...
    end a;
    procedure b (procedure x);
    var y: integer;

        procedure c ...
            ...
            y := ...;
            ...
        end c;
        x;
        b(c);
        ...
    end b;
    ...
    b(a);
    ...
end principal;
```

2) Co-Rotinas

Subprogramas não podem descrever unidades de programas que executam de maneira intercalada. Por exemplo, a simulação de um jogo de cartas com quatro jogadores não pode ser feita utilizando-se quatro sub-rotinas, uma para cada jogador, porque as sub-rotinas normalmente não guardam a história de sua execução e executam do seu início até o fim, antes de retornar à unidade que as chamou.

Assim, várias LP implementam o conceito de co-rotinas para atender tais necessidades. Distintamente de subprogramas, co-rotinas são unidades simétricas que explicitamente ativam outra unidade; elas não retornam, mas, em vez disso, passam o controle para outra unidade através de um comando do tipo *resume*. Quando uma co-rotina recebe o controle de outra unidade, ela normalmente só executa parcialmente. A sua execução é suspensa quando ela passa o controle para outra unidade e em um instante mais tarde sua execução pode ser reassumida, continuando normalmente a partir do ponto em que foi suspensa.

O modelo de execução utilizando uma pilha para armazenar os registros de ativação das unidades é inadequado para suportar o conceito de co-rotinas. Quando uma co-rotina “A” envia o comando *resume* para uma co-rotina “B”, o seu registro de ativação não pode ser desativado e, além disso, deve ser salvo o ponteiro para a instrução, após o comando *resume*, de onde essa co-rotina prosseguirá, quando o controle lhe for devolvido.

Cada co-rotina pode, por sua vez, chamar subprogramas; consequentemente, cada co-rotina precisa de uma pilha de registros de ativação, capaz de crescer e diminuir durante a sua execução, independentemente das pilhas das outras co-rotinas. Dessa forma, a criação de uma nova co-rotina corresponde à criação de uma nova pilha de execução. O modelo de execução pode ser visto como uma árvore de pilhas, onde a pilha do programa principal é a raiz da árvore e as pilhas das co-rotinas são seus filhos.

1.5.5. Gramáticas Livres de Contexto

Como já comentado anteriormente, a sintaxe de uma linguagem é descrita através de uma gramática, que consiste num conjunto de regras que definem todas as construções básicas que podem ser aceitas numa linguagem. Os elementos básicos de uma gramática são;

1. Conjunto de **símbolos terminais**: são os símbolos atômicos (não divisíveis) que podem ser combinados para formar construções válidas na linguagem. Símbolos terminais são geralmente um conjunto de caracteres através dos quais algumas linguagens podem considerar *strings* como símbolos.
2. Conjunto de **símbolos não terminais**: estes símbolos não estão incluídos na linguagem propriamente dita, mas são símbolos usados para representar definições intermediárias dentro da linguagem, definidas como produções. Os símbolos não terminais representam classes ou categorias sintáticas.
3. Conjunto de **produções**: consistem na definição de um símbolo não terminal. Possuem a forma "x ::= y", onde "x" é um símbolo não terminal e "y" é uma seqüência de símbolos que podem ser terminais ou não terminais.
4. **Símbolo inicial**: corresponde a um dos símbolos não terminais.

Duas regras devem ser obedecidas para estes elementos básicos formarem uma gramática: (1) cada símbolo não terminal deve aparecer no lado esquerdo de "::=" em pelo menos uma produção; (2) o símbolo inicial não deve aparecer no lado direito de "::=" em nenhuma produção [DER 90].

Para ilustrar o conceito de gramática, a seguir é apresentada a construção de uma gramática simples para descrever a linguagem de uma calculadora (já descrita em BNF na seção 1.5.1).

*Símbolos Terminais: 0 1 2 3 4 5 6 7 8 9 + - * / = .*
Símbolos Não Terminais: <calculation> <expression> <value> <sign>
<number> <unsigned> <digit> <operator>
Símbolo Inicial: <calculation>
Produções:

1. *<calculation> ::= <expression>=*
2. *<expression>::= <value>*
3. *<expression>::= <value> <operator> <expression>*
4. *<value> ::= <number>*
5. *<value> ::= <sign> <number>*
6. *<number>::= <unsigned>*
7. *<number>::= <unsigned> . <unsigned>*
8. *<unsigned> ::= <digit>*
9. *<unsigned> ::= <digit> <unsigned>*
10. *<digit>::= 0*
11. *<digit>::= 1*
12. *<digit>::= 2*
13. *<digit>::= 3*
14. *<digit>::= 4*
15. *<digit>::= 5*
16. *<digit>::= 6*
17. *<digit>::= 7*
18. *<digit>::= 8*
19. *<digit>::= 9*
20. *<sign>::= +*
21. *<sign> ::= -*
22. *<operator> ::= +*
23. *<operator> ::= -*
24. *<operator> ::= **
25. *<operator> ::= /*

Pode-se observar que esta gramática tem 16 símbolos terminais e 8 não terminais. Também observa-se que cada não terminal, com exceção do símbolo inicial, é definido por múltiplas produções. Apesar de que múltiplas produções não são requeridas, elas são comuns e representam definições alternadas. Produções recursivas, isto é, produções onde o não terminal que está sendo definido é encontrado na sua própria definição no lado direito da produção, são permitidas. Existem duas maneiras de se usar uma gramática como a apresentada. A primeira

consiste em gerar programas válidos na linguagem. Neste caso, começa-se pelo símbolo inicial e em cada passo substitui-se alguma definição por um não terminal, prosseguindo-se até que todos os símbolos restantes sejam símbolos terminais e gerando-se, assim, um programa válido. A seqüência abaixo ilustra estes procedimentos.

String Corrente	*Produção Aplicada*
<calculation>	*1*
<expression>=	*3*
<value> <operator> <expression>=	*4*
<number> <operator> <expression>=	*6*
<unsigned> <operator> <expression>=	*9*
<digit> <unsigned> <operator> <expression>=	*12*
2 <unsigned> <operator> <expression>=	*8*
2 <digit> <operator> <expression>=	*15*
25 <operator> <expression>=	*24*
25 <expression>=*	*2*
25 <value> =*	*4*
25 <number> =*	*7*
25 <unsigned> . <unsigned> =*	*8*
25 <digit> . <unsigned> =*	*11*
*25*1. <unsigned> =*	*8*
*25*1. <digit> =*	*15*
*25*1.5=*	

A segunda maneira na qual uma gramática pode ser usada é na redução de um programa válido para o símbolo inicial através da aplicação reversa de produções. Isto permite verificar que um *string* de símbolos terminais é de fato um programa na linguagem definida pela gramática. Esta derivação é somente possível se é feita uma escolha prudente da seqüência de produções a serem aplicadas, como mostra o exemplo da figura 1.7.

Como já comentado na seção 1.5.1, as gramáticas que podem ser descritas através de BNF ou EBNF são conhecidas como gramáticas livres de contexto. Isto significa que as definições válidas dos símbolos não terminais são independentes do contexto onde o símbolo é encontrado. A maioria das LP não podem ser completamente especificadas por uma gramática livre de contexto porque elas contêm algumas regras que são sensíveis ao contexto, o que significa que suas definições não terminais dependem do contexto. Por exemplo, a requisição comum de que uma variável deve ser declarada antes do seu uso não pode ser expressa em uma gramática livre de contexto, desde que a validade de uma variável depende se sua declaração está no contexto do seu uso ou não. Apesar de existirem ferramentas formais para expressar gramáticas sensíveis ao contexto, estas são muito mais complexas do que as livres de contexto e geralmente não são usadas para na especificação de linguagens. A abordagem comum consiste em especificar formalmente a parte livre de contexto da sintaxe de uma linguagem usando BNF ou uma ferramenta similar, e especificar informalmente a parte sensível ao contexto.

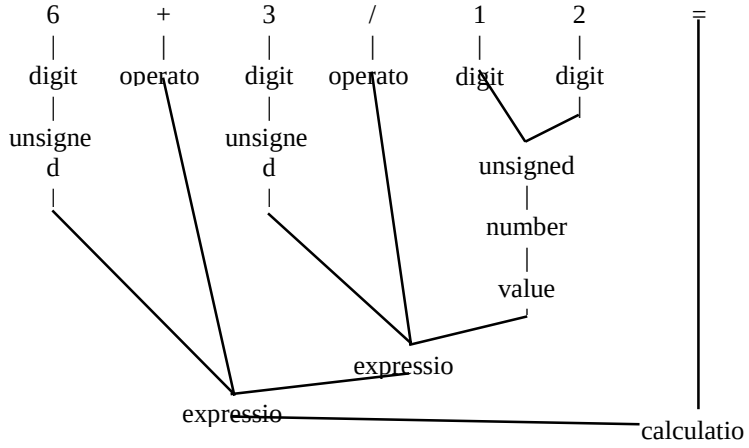

Figura 1.7 – Verificação que a expressão 6+3/12 é um calculo [DER 90]

Outro problema na especificação formal é a ambigüidade. A ambigüidade ocorre quando há mais de árvore de derivação associada com um *string* válido da linguagem. Para ilustrar, considera-se a gramática anterior modificada para trocar a produção "*expression*" por:

<*expression*> ::= <*value*> | <*expression*> <*operator*> <*expression*>

O resultado é uma gramática ambígua, como demonstrado na figura 1.8, onde há duas árvores de derivação válidas para o *string* 4+2*3.

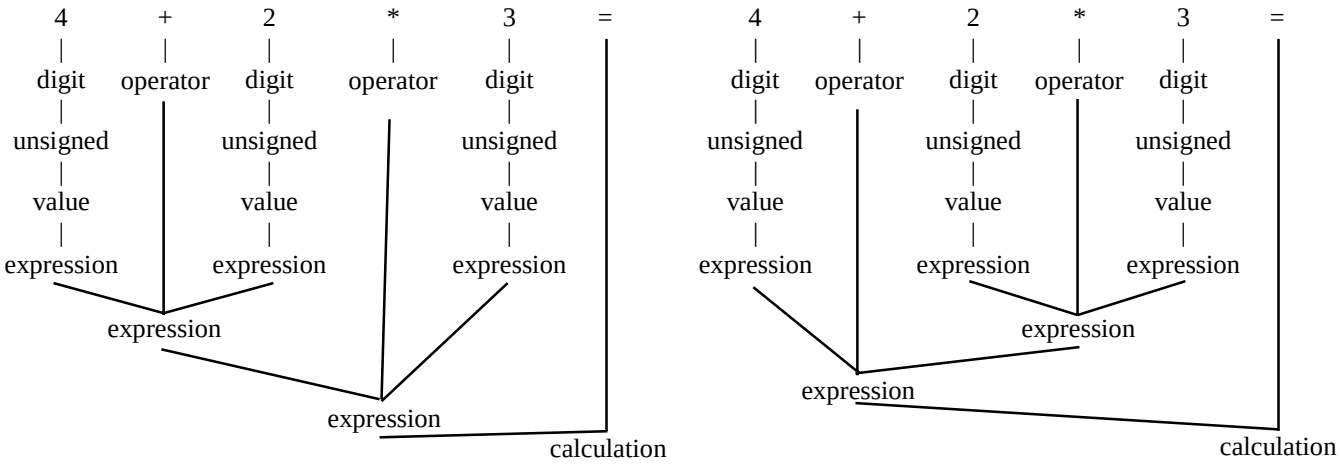

Figura 1.8 – Duas derivações para o *string* 4+2*3 [DER 90]

Se a gramática está sendo usada somente para determinar quando o *string* é válido ou não, esta ambigüidade não causa problemas, desde que qualquer derivação é suficiente para este objetivo. Freqüentemente, entretanto, uma gramática é usada para interpretar o significado de um *string*. Neste caso, as duas derivações diferentes possíveis têm significados diferentes no sentido que elas significam que os dois operadores são aplicados em direções opostas quando o comando é executado. Assim, a aplicação da árvore da direita representa um cálculo cujo resultado é 18, enquanto a árvore da esquerda representa um cálculo cujo resultado é 10.

Sendo assim, sempre que possível deve-se eliminar a ambigüidade. Como não há uma técnica genérica para eliminar a ambigüidade, cada situação deve ser cuidadosamente analisada. Algumas linguagens, entretanto, não podem ser representadas através de uma gramática não ambígua, e tentativas de encontrar tais representação são improdutivas. Tais linguagens são chamadas de linguagens ambíguas.