Unidade II

5 ANÁLISE SINTÁTICA (PARTE 2 – ASCENDENTE)

As configurações também serão da forma (α, y), em que α é o conteúdo da pilha e y é o resto da entrada. Entretanto, a convenção sobre o topo da pilha é invertida: o topo fica à direita, ou seja, o último símbolo de α é o símbolo do topo.

As mudanças de configuração são:

(1) **redução** pela regra $A \rightarrow \alpha$: permite passar da configuração ($\beta\alpha$, y) para a configuração (βA, y).

(2) **empilhamento**, ou deslocamento de um terminal s: permite passar da configuração (α, sy) para a configuração (as, y).

Observe que este último tipo de transição é o que permite trazer os terminais para a pilha a fim de que possam participar das reduções.

A configuração inicial para a entrada x é (ε, x), com a pilha vazia e, ao final, temos a configuração (S, ε), indicando que toda a entrada x foi lida e reduzida para S.

Na tabela abaixo é possível observar as configurações sucessivas de um analisador ascendente, para a cadeia x. A terceira coluna apresenta os passos correspondentes da derivação direita de x.

Pilha (topo à direita)	(Restante da) entrada	Derivação direita (invertida)
ε	a + a * a	a + a * a
a	+ a * a	
F	+ a * a	$\Leftarrow$ F + a * a
T	+ a * a	$\Leftarrow$ T + a * a
E	+ a * a	$\Leftarrow$ E + a * a
E +	a * a	
E + a	* a	
E + F	* a	$\Leftarrow$ E + F * a
E + T	* a	$\Leftarrow$ E + T * a
E + T *	A	
E + T * a	ε	
E + T * F	ε	$\Leftarrow$ E + T * F

E + T	ε	⇐ E + T
E	ε	⇐ E

Figura 10 - Correspondência entre as operações de um parser ascendente e da sequência invertida das ações realizadas durante o processo de derivação do não terminal mais à direita (right-most) para a cadeia a+a*a

5.1 Análise sintática sLR(1)

Um método de análise sintática ascendente que é chamado sLR(1). As letras que geram a sigla que dá nome indicam:

- S: a variante mais simples dos métodos LR(1).
- L: a cadeia de entrada é lida da esquerda para a direita (*left-to-right*).
- R: o método constrói uma derivação direita (*rightmost*) invertida.
- 1: apenas um símbolo do resto da entrada é examinado.

Como o símbolo inicial pode ocorrer em diferentes partes da árvore de derivação, devemos distinguir sua ocorrência relativa à raiz da árvore das demais. Para simplificar a identificação do término do processo de análise, acrescentamos à gramática uma nova regra inicial.

Com a introdução da regra S' → S, em que S é o símbolo inicial original e S' é um símbolo novo, temos o que chamamos de gramática aumentada e que passa a ter S' como seu o símbolo inicial. Assim, uma redução por essa regra indicará o fim da análise, já que S' nunca aparece à direita nas regras da gramática.

A gramática aumentada é usada na construção do analisador sLR(1) da gramática original. A construção desse tipo de analisador se baseia na formulação de um autômato de pilha que permitirá a identificação do momento adequado para realizarmos a redução dos símbolos da pilha segundo a regra apropriada. A construção desse autômato se dá por meio da descoberta dos estados a partir da análise das regras da gramática aumentada e da caracterização temporal realizada pelo parser.

Uma maneira conveniente para isso é a caracterização das regras na forma de itens. Um item $A \rightarrow \alpha \bullet \beta$ indica a possibilidade de que, no ponto atual em que se encontra a análise,

- A regra $A \rightarrow \alpha\beta$ foi usada na derivação da cadeia de entrada;
- Os símbolos terminais derivados a partir de α já foram todos encontrados;
- Faltam ainda encontrar os símbolos terminais derivados de β.

Assim, o marcador • indica o progresso da análise. Dessa maneira, um item $A \rightarrow \bullet\delta$ indica o início da busca por (uma cadeia derivada de) um δ, enquanto $A \rightarrow \delta\bullet$ indica o fim da busca pelos símbolos gerados δ, ou seja, o momento em que a redução de δ para A pode ser realizada.

Observe ainda que, em um dado momento, várias possibilidades precisam ser consideradas. Representamos um estado do processo de análise (estado do autômato) por um conjunto de itens. O estado inicial do processo de análise é dado pelo item $S' \rightarrow \bullet S$ proveniente da regra inicial e pode ser entendido como "só falta encontrar (uma cadeia derivada de) um S".

De acordo com as regras da gramática, quando um estado contém um item $A \rightarrow \alpha\bullet B\delta$ é necessário acrescentar a cada estado as possibilidades correspondentes, ou seja, os itens correspondentes às regras de B para que seja possível dirigir a busca por B. Esses são os itens da forma $B \rightarrow \bullet\gamma$, para todas as regras $B \rightarrow \gamma$. Esse processo é repetido enquanto for necessário, sendo denominado de fechamento do estado. O estado inicial é dado pelo fechamento de $\{ S' \rightarrow \bullet S \}$.

Vejamos um exemplo de caso, para isso considere a gramática:

G = ({E, T, F}, {a, +, *, (,)}, P, E)
P = {
1. E → E + T ,
2. E → T ,
3. T → T * F ,
4. T → F ,
5. F → (E) ,
6. F → a }

A gramática aumentada passaria a ser dada na seguinte forma:

G = ({S', E, T, F}, {a, +, *, (,)}, P, S')
P = {
0. S' → E ,
1. E → E + T ,
2. E → T ,
3. T → T * F ,
4. T → F ,
5. F → (E) ,
6. F → a }

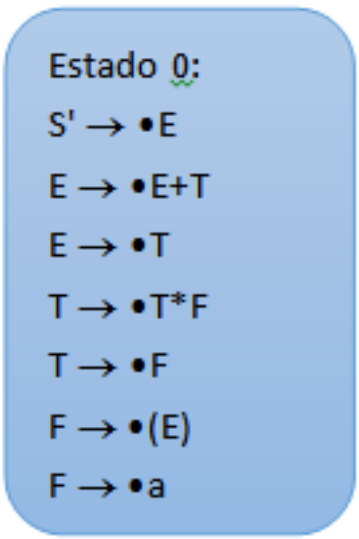
Os itens possíveis para a gramática aumentada serão finitos, uma vez que a quantidade de regras sempre é finita. Nesse caso, compreendem o total de 20, sendo: $S' \rightarrow \bullet E$, $S' \rightarrow E\bullet$, $E \rightarrow \bullet E+T$, $E \rightarrow E\bullet+T$, $E \rightarrow E+\bullet T$, $E \rightarrow E+T\bullet$, ..., $F \rightarrow \bullet a$, $F \rightarrow a\bullet$

A construção do autômato tem por início a determinação do estado inicial que é dado o fechamento de $\{ S' \rightarrow \bullet E \}$. Esse item é entendido como "*estamos aguardado os símbolos derivados a partir de E*".

Como E é um símbolo não terminal, isso implica que estamos aguardando qualquer coisa possível de ser derivada a partir de E, então devemos acrescentar outros itens nesse estado, de modo que representem essa situação; ou seja, devemos acrescentar os itens E → •E+T e E → •T para que representem o nosso momento do processo de derivação.

Na prática podemos dizer que como há um ponto antes de E, devemos acrescentar itens referentes a todas as possibilidades de derivação a partir de E. Ainda nesse estado e seguindo o mesmo raciocínio, por causa do ponto que aparece antes de T no item E → •T, devemos acrescentar os itens relativos às possíveis derivações a partir de T; ou seja, T → •T*F e T → •F. Novamente, por causa do ponto antes de F, acrescentamos F → •(E) e F → •a.

Como não há mais qualquer situação em que o nosso marcador preceda um símbolo não terminal, encerramos o processo de fechamento do estado. Assim, o estado inicial (estado 0) será, então, composto pelos itens:



À medida que a análise prossegue, o ponto deve caminhar para a direita nos diversos itens do estado. Se for encontrado um símbolo X, passamos de um item A → α•Xεpara um item A → αX•ε Note que os símbolos terminais serão encontrados na entrada; enquanto os símbolos não terminais serão encontrados como resultado de reduções.

Se o autômato de reconhecimento estiver em um estado *p* que tem itens com a marcação (ponto) antes de um símbolo X, uma transição rotulada com este símbolo X nos levará a outro estado *q* que terá um item com a marcação depois da ocorrência de X (para cada item do estado *p* que apresente o ponto antes de X). Os outros itens do estado *q* serão obtidos a partir do fechamento do estado. Considerando o estado 0 do nosso exemplo, teríamos:

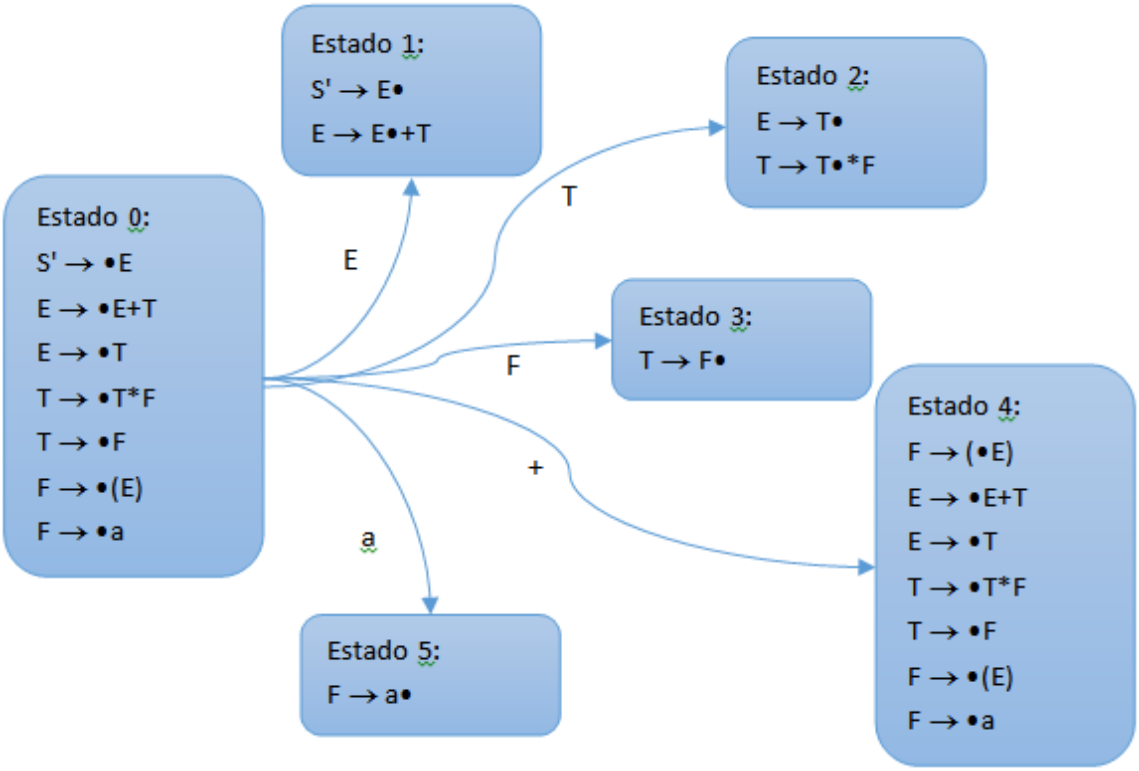


Figura 11 - Análise das possíveis transições a partir do estado 0 e a geração de novos estados

Para gerar a tabela de transições utilizadas pelo analisador, geramos todos os estados e as transições possíveis, começando pelo estado inicial e depois para cada um dos outros estados gerados a partir dele. O número de estados desse autômato sempre será finito, uma vez que o número de itens é finito.

A coleção completa de estados para a gramática que foi dada no exemplo é a seguinte:

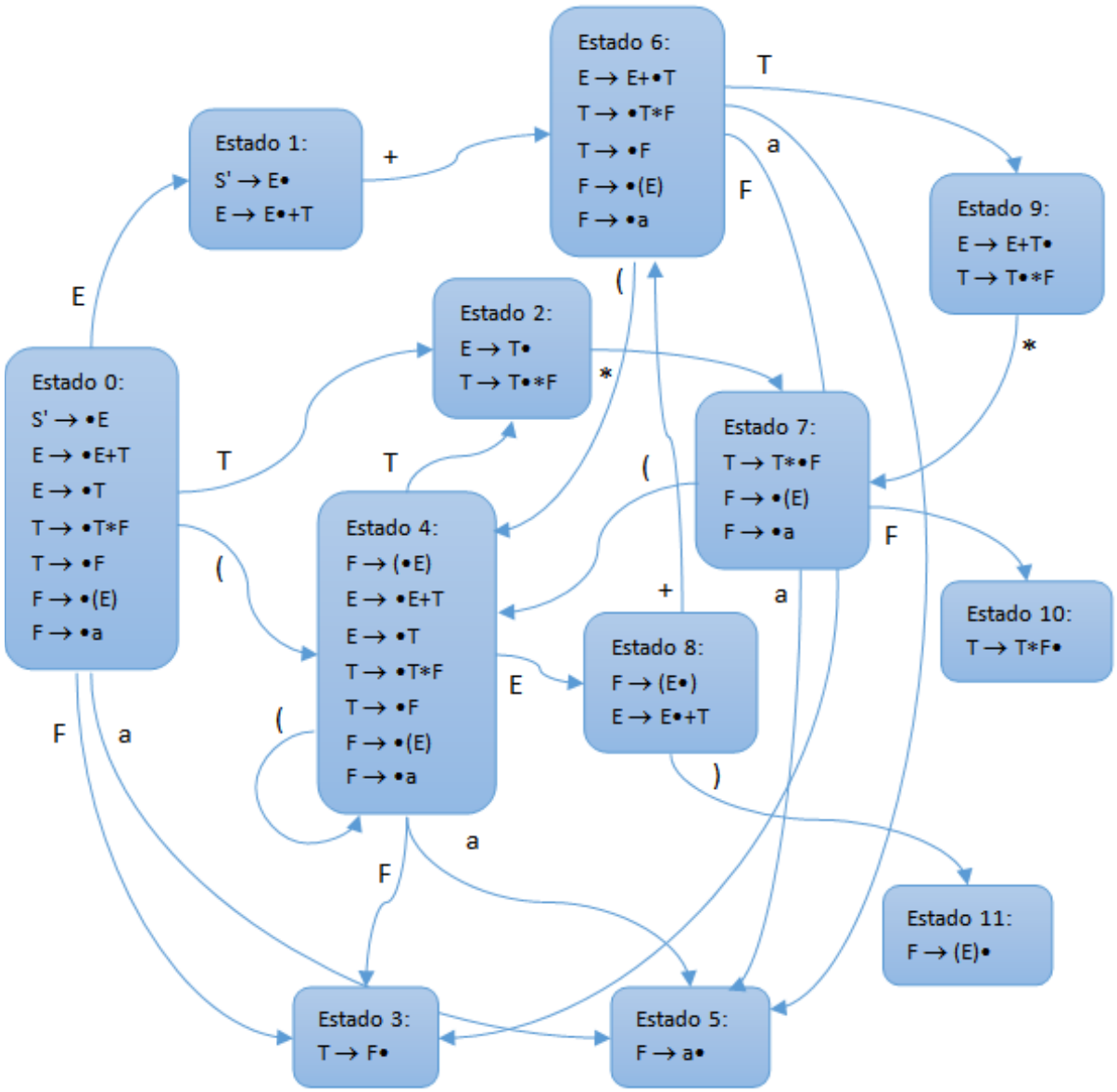


Figura 12 - Autômato de pilha completo para construção do analisador ascendente

5.2 Montando a tabela de movimentos do analisador

Uma vez estabelecido o autômato de pilha (dado a partir da gramática da linguagem), podemos então criar a tabela de movimentos que definirá as ações do analisador. As transições serão mapeadas como entradas da tabela, tendo como índices o número que representa o estado e o símbolo que rotula a transição.

Desta forma, a tabela deverá conter tantas linhas quantos sejam o número de estados e tantas colunas quanto sejam os diferentes símbolos que possam gerar transições entre os estados. Adicionalmente, haverá ainda uma coluna correspondente ao símbolo que representa o fim de arquivo (representado por $ ou *EOF – End Of File*). Note que a tabela inclui alguns estados a partir dos quais não há transições,

como no caso dos estados 3, 5, 10 e 11.

Considerando o autômato dado na figura 12, a tabela assumirá a seguinte configuração:

	E	T	F	(	A	+	*	)	$
0	1	2	3	4	5	-	-	-	-
1	-	-	-	-	-	6	-	-	-
2	-	-	-	-	-	-	7	-	-
3	-	-	-	-	-	-	-	-	-
4	8	2	3	4	5	-	-	-	-
5	-	-	-	-	-	-	-	-	-
6	-	9	3	4	5	-	-	-	-
7	-	-	10	4	5	-	-	-	-
8	-	-	-	-	-	6	-	11	-
9	-	-	-	-	-	-	7	-	-
10	-	-	-	-	-	-	-	-	-
11	-	-	-	-	-	-	-	-	-

Figura 13 - Tabela de movimentos do parser LR(1) – parcial

O analisador sLR(1) é um analisador ascendente. Em vez de símbolos, a pilha do analisador contém os estados correspondentes aos símbolos. Primeiro, observamos que a cada estado *q*, com exceção do estado inicial, corresponde exatamente um símbolo X, que é o único símbolo que ocorre depois do ponto, nos itens do estado *q*. Todas as transições para *q* são feitas com o símbolo X, podendo ocorrer, entretanto, que dois ou mais estados sejam acessíveis pelo mesmo símbolo X. Nesse caso, os estados se distinguem por conter informação adicional sobre a posição em que o símbolo X ocorre na cadeia de entrada.

As duas ações possíveis em analisadores ascendentes se aplicam aqui. Um empilhamento (*shift*) pode ocorrer quando existe uma transição com um terminal a partir do estado corrente (o estado do topo da pilha).

Quando existe um item completo $B \rightarrow \gamma\bullet$, no estado corrente, pode ser feita uma redução (*reduce*) pela regra $B \rightarrow \gamma$. Em um analisador sLR(1), a regra é: "*reduza pela regra* $B \rightarrow \gamma$ *se o símbolo da entrada pertencer ao Follow(B)*".

O primeiro símbolo pertencente ao restante da entrada é conhecido como o símbolo de *lookahead* e é com base nele e no estado corrente que se determinam as ações do parser.

A tabela para o analisador sLR(1) pode conter as seguintes ações, dadas em função do estado *q* (do topo da pilha) e do símbolo *s* de *lookahead* (dado pela entrada):

Empilhamento: o empilhamento do estado *p* (que representa o símbolo *s*) deve ser empilhado e o analisador léxico deve ser acionado para obter outro símbolo da entrada.

Redução: se T[*q*, *s*] = reduce B → γ, os |γ| estados correspondentes a devem ser retirados da pilha e o estado δ(*q*, B) deve ser empilhado, representando B.

Aceitação: se T[*q*, *s*] = reduce S' → S, o processo se encerra com sucesso.

Nos exemplos, uma ação de empilhamento (*shift*) *q* será representada apenas pelo número do estado *q*; e uma ação de redução (*reduce*) B → γ, será representada por rN, em que N é o número da regra B → γ. Desta forma, a ação de aceitação e parada será indicada como r0.

Completando as informações dadas na figura 13 com as operações de redução, a tabela de ações do parser finalizada ficará da seguinte maneira:

	E	T	F	(	a	+	*	)	$
0	1	2	3	4	5	-	-	-	-
1	-	-	-	-	-	6	-	-	r0
2	-	-	-	-	-	r2	7	r2	r2
3	-	-	-	-	-	r4	r4	r4	r4
4	8	2	3	4	5	-	-	-	-
5	-	-	-	-	-	r6	r6	r6	r6
6	-	9	3	4	5	-	-	-	-
7	-	-	10	4	5	-	-	-	-
8	-	-	-	-	-	6	-	11	-
9	-	-	-	-	-	r1	7	r1	r1
10	-	-	-	-	-	r3	r3	r3	r3
11	-	-	-	-	-	r5	r5	r5	r5

Figura 14 - Tabela de movimentos do parser LR(1) – completa

Para entendermos o uso dessa tabela e, consequentemente, o funcionamento do parser, vejamos como ficaria a análise do programa (a + a) * a.

Na configuração inicial, a pilha contém apenas o estado inicial 0 e a entrada contém a cadeia completa. À medida que transcorre a análise, acrescentamos à pilha os estados correspondentes às transições grafadas na tabela. Neste exemplo, além dos estados, inserimos na pilha ainda os símbolos (grafados em azul) correspondentes às transições e que devem ser entendidos apenas como comentários de propósitos didáticos.

Os passos para o processamento da cadeia são:

Pilha	Entrada	Ação
0	(a + a) * a	Empilhar o estado 4
0 (4	a + a) * a	Empilhar o estado 5
0 (4 a5	+ a) * a	Reduzir usando a regra 6 Note que o autômato retornará ao estado 4 após a retirada do símbolo *a* da pilha para que ocorra a redução por F.
0 (4 F3	+ a) * a	Reduzir usando a regra 4
0 (4 T2	+ a) * a	Reduzir usando a regra 2
0 (4 E8	+ a) * a	Empilhar o estado 6
0 (4 E8 +6	a) * a	Empilhar o estado 5
0 (4 E8 +6 a5	) * a	Reduzir usando a regra 6
0 (4 E8 +6 F3	) * a	Reduzir usando a regra 4
0 (4 E8 +6 T9	) * a	Reduzir usando a regra 1 Observe que três elementos são removidos da pilha para que ocorra a redução por E ε E+T. Com a retirada, o autômato assume o estado 4 e após a inserção do não terminal E assume o estado 8.
0 (4 E8	) * a	Empilhar o estado 11
0 (4 E8)11	* a	Reduzir usando a regra 5
0 F3	* a	Reduzir usando a regra 4
0 T2	* a	Empilhar o estado 7
0 T2 *7	a	Empilhar o estado 5
0 T2 *7 a5	ε	Reduzir usando a regra 6
0 T2 *7 F10	ε	Reduzir usando a regra 3
0 T2	ε	Reduzir usando a regra 2
0 E1	ε	Reduzir usando a regra 0 (aceitar)

Figura 15 - Sequência de ações realizadas pelo parser sLR(1) para a validação da cadeia (a+a)*a

Como se pode observar, a sequência das reduções foi 6 4 2 6 4 1 5 4 6 3 2 0, tal qual era esperado e que correspondente à derivação direita S' εE εT εT*F εT*a εF*a ε(E)*a ε(E+T)*a ε(E+F)*a ε(E+a)*a ε(T+a)*a ε(F+a)*a ε(a+a)*a

Leituras recomendadas:

Capítulo 4, seção 4.5 do livro "Compiladores: Princípio, Técnicas e Ferramentas" (livro do Dragão)

Capítulo 3, seções 3.3 e 3.4, do livro "Implementação de Linguagens de Programação: Compiladores"

6 ANÁLISE SEMÂNTICA

As técnicas de análise sintática vistas até o momento descrevem, basicamente, elementos reconhecedores; ou seja, máquinas que determinam se uma cadeia (programa fonte) pertence ou não à linguagem reconhecida.

Observe que, no caso de aceitação, a informação sobre a forma de derivação de *x* deve ser usada na geração do código para *x*; e, no caso de não aceitação, a informação disponível deve ser usada para tratamento de erros, sinalizando (para que o programador possa corrigi-lo) a recuperação (para que o parser possa continuar a análise).

Além da informação sintática sobre a forma de derivação da entrada, deve ficar disponível também a informação léxica, composta essencialmente pelas cadeias correspondentes aos chamados *tokens* variáveis (identificadores, literais inteiros, reais, cadeias ...). Essa informação deve ser processada nas fases seguintes do compilador.

Uma técnica geral de tratamento das informações obtidas durante as fases de análise léxica e sintática são as baseadas em gramáticas de atributos. Utilizando esse tipo de gramática, é possível especificar qualquer forma de tradução dirigida pela sintaxe, associando valores – os atributos – aos símbolos terminais e não terminais de uma gramática livre de contexto.

6.1 Tarefas do analisador semântico

O analisador semântico é responsável fundamentalmente por realizar três tarefas:

1. Construir uma descrição interna de todos tipos e estruturas de dados que foram definidos pelo programador. Esses elementos estendem as definições da própria linguagem, criando novos tipos de dados e possivelmente definindo relações entre eles;

2. Armazenar em uma estrutura chamada Tabela Símbolos as informações sobre cada um dos identificadores que são usados no programa, ou seja, registrar identificadores de constante, tipos, variáveis, procedimentos, parâmetros e funções. Vale lembrar que tais elementos, eventualmente, podem ter sido definidos em outros arquivos e apenas utilizados no código corrente.

3. Verificar o programa quanto a erros semânticos (erros dependentes de contexto) e checagens de tipos com base nas informações contidas na tabela de símbolos.

6.2 Os componentes semânticos

As várias tarefas realizadas pelo analisador semântico podem ser relacionadas a dois momentos diferentes de um programa: o primeiro, ligado ao tempo de compilação; e o segundo, relacionado com a execução do programa. A esses dois diferentes momentos estão ligados os **componentes semânticos estático** e **componentes de tempo de execução**.

O componente estático está preocupado com as tarefas ligadas a verificação do uso adequado dos identificadores na construção do programa. Essas atividades compreendem tarefas ligadas a análise contextual, tais como declarações prévias de variáveis e escopo de uso, por exemplo; além da checagem de tipos e compatibilidade entre variáveis, expressões e chamadas de sub-rotinas. Forma um conjunto de restrições que determinam se programas sintaticamente corretos também são válidos quanto a aspectos semânticos.

As atividades compreendidas pelo componente semântico estático são:

- Checagem de tipos;
- Análise de escopo de declarações;
- Verificação da quantidade e dos tipos dos parâmetros em sub-rotinas.

As definições dessa natureza para uma linguagem de programação podem ser formalmente especificadas por uma gramática de atributos.

Há, ainda, outros aspectos relacionados à semântica e que estão ligados à maneira como o programa se comportará durante sua execução. Essas questões estão relacionadas ao componente semântico de tempo de execução.

Fundamentalmente são definições utilizadas para especificar o que o programa faz, ou seja, estabelecem relações entre o programa-fonte (objeto estático) e a sua execução dinâmica (comportamento).

Para ilustrar uma dessas questões, considere, por exemplo, o seguinte comando:

if (i<>0) &t&t (K/I > 10) ...

O destino do desvio é determinado pela avaliação da expressão lógica dada no comando e cujo resultado depende dos resultados parciais de duas cláusulas combinadas pelo operador lógico de conjunção (AND). Do ponto de vista lógico, devemos considerar que se a primeira cláusula for falsa não haveria a necessidade de verificarmos a segunda. Porém, se esta chegar a ser avaliada incorrerá em um erro de divisão por zero.

Nesse exemplo, o programador dispôs as cláusulas na ordem de avaliação correta para que a segunda seja avaliada apenas depois de confirmado que a variável i não tem valor igual a zero. Isso pode ser assumido por ele porque a linguagem estabelece uma regra semântica que diz que a segunda cláusula não será avaliada se a primeira for falsa, quando o contexto envolve um operador de conjunção.

A especificação dos aspectos de tempo de execução em uma linguagem é importante para a etapa de geração de código. Geralmente essas questões são especificadas de maneira informal, mas também é possível o uso de formalismos, tais com as gramáticas de atributos.

6.3 Gramáticas de atributos

Uma gramática de atributos é composta de duas partes:

Uma gramática livre de contexto, que descreve uma linguagem;

Regras de cálculo, que permitem calcular valores que estão associados aos símbolos terminais e não terminais da gramática.

Vejamos, por exemplo, uma gramática de atributos abaixo, que permite calcular o valor em decimal de um número binário com parte fracionária:

$G = (\{N, B, H\}, \{0, 1, .\}, P, N)$

$P = \{$

1. $N \rightarrow H_1 . H_2$ — $N.v := H_1.v + H_2.v$; $H_1.p := 0$; $H_2.p := -H_2.c$,
2. $N \rightarrow H$ — $N.v := H.v$; $H.p := 0$,
3. $H_0 \rightarrow H_1 B$ — $H_0.v := H_1.v + B.v$; $H_0.c := H_1.c + 1$; $H_1.p := H_0.p + 1$; $B.p := H_0.p$,
4. $H \rightarrow B$ — $H.v := B.v$; $H.c := 1$; $B.p := H.p$,
5. $B \rightarrow 0$ — $B.v := 0$,
6. $B \rightarrow 1$ — $B.v := 2^{B.p}$ $\}$

Note que algumas ocorrências de símbolos na gramática estão diferenciadas por índices inferiores (subescritos). Isso nos permite fazer referência ao "primeiro H" e ao "segundo H" da primeira regra, por exemplo. A numeração das regras de produção também serve apenas para facilidade de referência.

As regras de cálculo permitem calcular os atributos *v* (valor), *p* (posição) e *c* (comprimento). Note que em N só há referências ao valor (N.v), em B, ao valor e à posição (B.v e B.p), mas em H existem referências aos três atributos (H.v, H.p, H.c).

Para cada regra sintática há regras semânticas que permitem o cálculo de alguns atributos e que em algumas vezes são determinados em função de outros atributos. Por exemplo, para a regra 1, temos (entre outras) a regra semântica:

$N.v := H_1.v + H_2.v$

.. que indica que "*o valor de N é obtido somando os valores do primeiro H e do segundo H*".

Essas regras, usualmente, são criadas por um processo de programação, porém esse processo não especifica a sequência em que as regras semânticas devem ser aplicadas. O momento para calcular N.v não é especificado e a aplicação pode ser feita a qualquer momento, desde que $H_1.v$ e $H_2.v$ tenham sido calculados antes. Dizemos que N.v depende de $H_1.v$ e $H_2.v$.

Fundamentalmente, as regras podem ser aplicadas em qualquer ordem, desde que os valores usados nas regras tenham sido calculados antes, ou seja, obedecendo a relações de dependência entre atributos.

Suponha que estejamos processando a cadeia 101.011. Com base na gramática do exemplo, construímos a árvore sintática para a cadeia utilizando um analisador ascendente, que resultaria em:

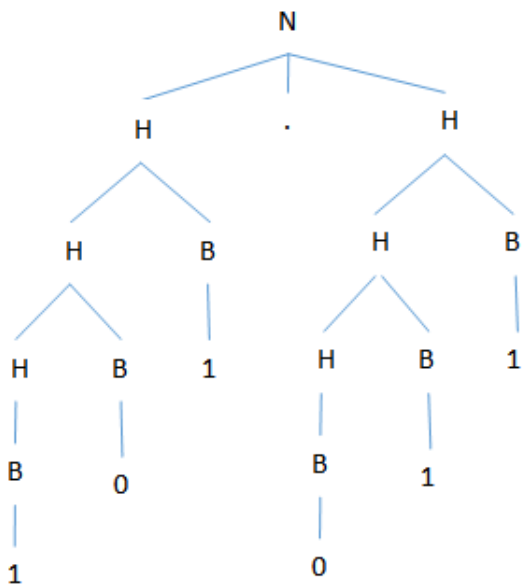


Figura 16 - Árvore sintática para a cadeia 101.011

6.4 Categoria dos atributos

Podemos classificar os atributos em herdados e sintetizados, de acordo com símbolo da gramática a que está associado. Essa classificação se baseia na posição do símbolo na regra e atende aos seguintes critérios:

Se tivermos uma regra r: $A \rightarrow X_1X_2...X_m$ e uma regra semântica associada à regra r permite calcular um atributo x de A (A.x := ...), então dizemos que o atributo x de A é **sintetizado**.

Por outro lado, se na regra r: $A \rightarrow X_1X_2...X_m$, podemos calcular um atributo y de qualquer um dos símbolos X_i, (X_i.y := ...), o atributo y de X_i é dito **herdado**.

Desta maneira, a ordem de cálculo não é arbitrária. Como o lado esquerdo da regra aparece em cima na árvore de derivação, em geral, os atributos sintetizados são calculados em função de outros atributos correspondentes a nós inferiores na árvore e, portanto, valores sucessivos do atributo podem ser calculados subindo na árvore. De maneira análoga, os valores sucessivos de atributos herdados são obtidos dos elementos superiores e podem ser calculados descendo na árvore.

Para maior clareza, façamos algumas restrições:

1. Cada atributo deve ter apenas uma classificação, sendo sintetizado ou herdado, isto é, um atributo a de um símbolo X não pode ser calculado, uma vez quando o símbolo X se encontra do lado esquerdo de uma regra, e outra vez quando X se encontra do lado direito de outra regra (observe que os atributos são associados aos símbolos e, assim, podemos ter um atributo herdado X.a, e outro atributo Y.a sintetizado).

2. Todas as regras devem ser apresentadas regras semânticas para calcular todos os atributos associados a um símbolo, isto é, em uma regra $A \rightarrow X_1X_2...X_m$ devem estar presentes regras semânticas para cálculo de todos os atributos sintetizados de A e de todos os atributos herdados de todos os X_i.

Uma gramática que satisfaz essas duas restrições é chamada uma **gramática normal**.

Retomando nosso exemplo, analisamos que os atributos N.v, H.v, H.c, B.v são sintetizados e os atributos H.p e B.p são herdados. Observe que por meio da regra (3) estão calculados os atributos sintetizados de H_0 (H_0.v e H_0.c), os atributos herdados de H_1 (H_1.p) e de B (B.p).

Assim, a ordem de cálculo que utilizaremos para a tabela será dada em três etapas:

- *c* (sintetizado) será calculado primeiro, subindo a árvore;
- *p* (herdado) será calculado logo depois, descendo a árvore;
- *v* (sintetizado) será calculado por último, subindo a árvore.

A ordem *c*, *p*, *v* é usada porque nenhum valor de *c* depende de algum valor de outro atributo, mas valores de *p* podem depender dos valores de *c* (na regra 1, temos $H_2.p := -H_2.c$) e valores de v podem depender dos valores de p (na regra 6, temos $B.v := 2^{B.p}$).

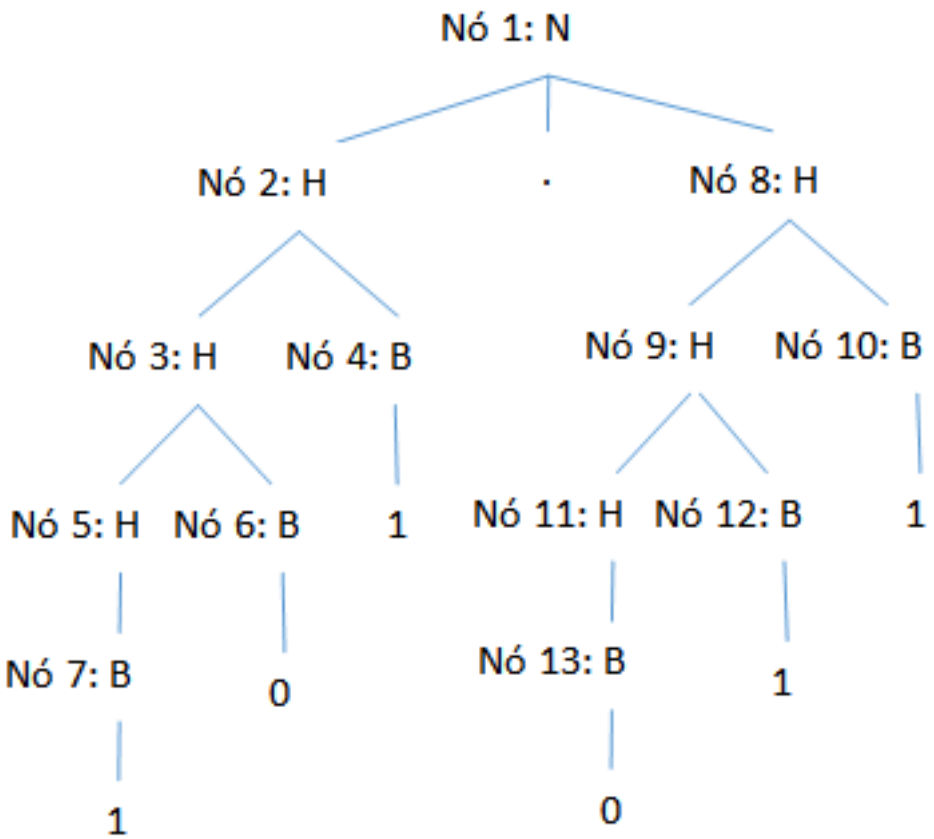


Figura 17 - Árvore sintática da sentença 101.011 com cada um de seus nós numericamente identificados

A tabela dada pela figura 18 apresenta os passos para o cálculo dos atributos para o exemplo dado anteriormente.

Nó / atrib	Nó / regra	Nós envolvidos	Cálculo
5 / c	5 / 4	5(H), 7(B)	H.c := 1
11 / c	11 / 4	11(H), 13(B)	H.c := 1
3 / c	3 / 3	$3(H_0)$, $5(H_1)$, 6(B)	$H_0.c := H_1.c + 1 = 2$
9 / c	9 / 3	$9(H_0)$, $11(H_1)$, 12(B)	$H_0.c := H_1.c + 1 = 2$
2 / c	2 / 3	$2(H_0)$, $3(H_1)$, 4(B)	$H_0.c := H_1.c + 1 = 3$
8 / c	8 / 3	$8(H_0)$, $9(H_1)$, 10(B)	$H_0.c := H_1.c + 1 = 3$
2 / p	1 / 1	1(N), $2(H_1)$, $8(H_2)$	$H_1.p := 0$
8 / p	1 / 1	1(N), $2(H_1)$, $8(H_2)$	$H_2.p := -H_2.c = -3$
3 / p	2 / 3	$2(H_0)$, $3(H_1)$, 4(B)	$H_1.p := H_0.p + 1 = 1$
4 / p	2 / 3	$2(H_0)$, $3(H_1)$, 4(B)	$B.p := H_0.p = 0$
9 / p	8 / 3	$8(H_0)$, $9(H_1)$, 10(B)	$H_1.p := H_0.p + 1 = -2$

10 / p	8 / 3	$8(H_0)$, $9(H_1)$, 10(B)	B.p := H_0.p = -3
5 / p	3 / 3	$3(H_0)$, $5(H_1)$, 6(B)	H_1.p := H_0.p + 1 = 2
6 / p	3 / 3	$3(H_0)$, $5(H_1)$, 6(B)	B.p := H_0.p = 1
11 / p	9 / 3	$9(H_0)$, $11(H_1)$, 12(B)	H_1.p := H_0.p + 1 = -1
12 / p	9 / 3	$9(H_0)$, $11(H_1)$, 12(B)	B.p := H_0.p = -2
7 / p	5 / 4	5(H), 7(B)	B.p := H.p = 2
13 / p	11 / 4	11(H), 13(B)	B.p := H.p = -1
7 / v	7 / 6	7(B)	B.v := $2^{B.p}$ = 4
13 / v	13 / 5	13(B)	B.v := 0
5 / v	5 / 4	5(H), 7(B)	H.v := B.v = 4
6 / v	6 / 5	6(B)	B.v := 0
11 / v	11 / 4	11(H), 13(B)	H.v := B.v = 0
12 / v	12 / 6	12(B)	B.v := $2^{B.p}$ = ¼
3 / v	3 / 3	$3(H_0)$, $5(H_1)$, 6(B)	H_2.v := H_1.v + B.v = 4
4 / v	4 / 6	4(B)	B.v := $2^{B.p}$ = 1
9 / v	9 / 3	$9(H_0)$, $11(H_1)$, 12(B)	H_2.v := H_1.v + B.v = ¼
10 / v	10 / 6	6(B)	B.v := $2^{B.p}$ = 1/8
2 / v	2 / 3	$2(H_0)$, $3(H_1)$, 4(B)	H_2.v := H_1.v + B.v = 5
8 / v	8 / 3	$8(H_0)$, $9(H_1)$, 10(B)	H_2.v := H_1.v + B.v = 3/8
1 / v	1 / 1	1(N), $2(H_1)$, $8(H_2)$	N.v := H_1.v + H_2.v = 5,3/8

Figura 18 - Cálculo dos atributos v, p e c para a cadeia 101.011

6.5 Gramáticas S-Atribuídas

Uma definição dirigida pela sintaxe somente com atributos sintetizados é chamada de definição S-atribuída. Na tradução dirigida pela sintaxe assume-se que os terminais tenham somente atributos sintetizados na medida em que as definições não providenciem quaisquer regras semânticas.

Exemplo de uma gramática desse tipo é dado a seguir:

G = ({E, T, F}, {a, +, *, (,)}, P, E)

P = {
1. $E_0 \rightarrow E_1 + T$ E_0.val := E_1.val + T.val ,
2. $E \rightarrow T$ E.val := T.val ,
3. $T_0 \rightarrow T_1 * F$ T_0.val := T_1.val * F.val ,
4. $T \rightarrow F$ T.val := F.val ,
5. $F \rightarrow (E)$ F.val := E.val ,
6. $F \rightarrow a$ F.val := id.*lexval* }

6.6 Gramáticas L-Atribuídas

Atributos sintetizados são bastante usados na prática, entretanto o uso de atributos herdados é conveniente para expressar construções de Ling. de Programação em relação ao contexto em que aparecem.

Gramáticas desse tipo costumam ser aplicadas para a verificação de tipos ou, ainda, para controlar se um identificador aparece do lado esquerdo (endereço) ou direito (valor) de uma atribuição. Uma gramática desse tipo é dada a seguir:

G = ({D, T, L}, {id, int, float}, P, D)

P = {

1.	$D \rightarrow T\ L$	L.in := T.tipo	,
2.	$T \rightarrow int$	T.tipo := *inteiro*	,
3.	$T \rightarrow float$	T.tipo := *real*	,
4.	$L_0 \rightarrow L_1$, id	L_1.in := L_0.in	
		addTbSimbolos(id.token, L_0.in)	,
5.	$L \rightarrow id$	addTbSimbolos(id.token, L.in)	}

Leituras recomendadas:

Capítulo 5, seções 5.1, 5.2, 5.3 e 5.4, do livro "Compiladores: Princípio, Técnicas e Ferramentas" (livro do Dragão)

Capítulo 4, seções 4.1, 4.2 e 4.3, do livro "Implementação de Linguagens de Programação: Compiladores"

Exercícios resolvidos:

1. Apesar de a maioria das linguagens de programação de alto nível exibir diversos tipos de dependências de contexto, elas (as linguagens), normalmente, são representadas, sintaticamente, por meio de gramáticas livres de contexto. Justifique:

a) O motivo de se usar essa estratégia;

Resp.: Formalismos para representar dependências de contexto são, geralmente, mais complexos e trabalhosos de se usar do que aqueles usados para representar linguagens livres de contexto.

b) Quais as consequências práticas da estratégia na especificação da linguagem-fonte?

Resp.: A linguagem especificada por meio de uma gramática livre de contexto é mais ampla do que a linguagem desejada.

c) Quais as consequências práticas dela no desenvolvimento do compilador para a linguagem?

Resp.: Torna-se necessário introduzir uma fase de análise de contexto posterior à análise livre de contexto, para detectar eventuais erros de contexto contidos no programa-fonte.

2. Responda as questões a seguir:

a) Em que consiste a subfase de identificação durante a fase de análise de contexto?

Resp.: Consiste na vinculação entre todas as referências e todas as declarações de nomes. Essa vinculação é feita com base nas regras de escopo da linguagem.

b) Qual a importância da tabela de símbolos durante essa subfase?

Resp.: A tabela de símbolos (ou de identificação) é uma estrutura de dados fundamental para permitir que a identificação ocorra em um único passo (no máximo dois, para linguagens que permitem referências para a frente). Ela serve para armazenar todos os nomes declarados e seus respectivos atributos. A busca é sempre feita na tabela.

c) Quais as operações básicas que devem ser prevista pela tabela de símbolos para suportar a identificação de linguagens com estrutura de blocos aninhados?

Resp.: Inserção de nomes (enter), busca de nomes (retrieve), início de escopo (open) e fim de escopo (close).

d) Que tipo de erros são reportados durante a subfase de identificação?

Resp.: Nomes não declarados e nomes declarados mais de uma vez no mesmo bloco.

e) Em que consiste a subfase de verificação de tipos durante a fase de análise de contexto?

Resp.: Consiste na verificação da adequação dos tipos dos operandos aos tipos requeridos pelos operadores utilizados.

f) De que maneira estão relacionadas as subfases de identificação e de verificação de tipos?

Resp.: A subfase de verificação de tipos depende da subfase de identificação, pois é por meio da vinculação do nome com a respectiva declaração que se obtém o tipo do operando e se pode efetuar a verificação da assinatura da operação.

g) Que tipo de erros são reportados durante a subfase de verificação de tipos?

Resp.: Tipos incompatíveis com a operação em questão.

7 GERAÇÃO DE CÓDIGO E OTIMIZAÇÃO

Segundo o modelo de Análise e Síntese, podemos considerar que os módulos iniciais do compilador (*front-end modules*) traduzem um programa fonte para uma representação intermediária, enquanto seus módulos finais (*back-end modules*) geram o código objeto final.

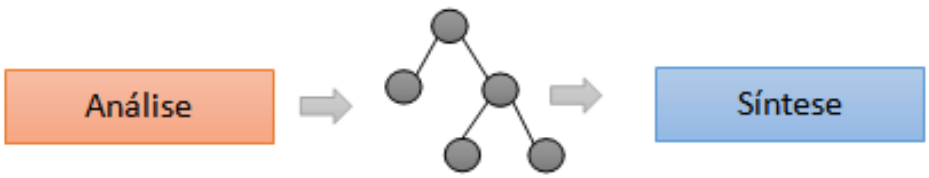


Figura 19 - Modelo Análise e Síntese

Desta forma, a fase de Síntese tem por objetivo geral produzir o código do alvo a partir do produto resultante da fase de análise, isto é, partir da estrutura criada na análise e construir uma sequência de instruções que seja adequada e equivalente em termos funcionais.

Essa tarefa é, geralmente, dividida em duas etapas:

- Na primeira, ocorre a tradução da estrutura construída na análise sintática para um código em linguagem intermediária, usualmente independente do processador.

- Em seguida é realizada a tradução do código em linguagem intermediária para a linguagem simbólica do processador-alvo.

Vale ressaltar que a produção efetiva do código binário não compete ao compilador, sendo realizada por outro programa chamado montador. Na prática, esses limites são um pouco obscuros, uma vez que muitos fabricantes criam compiladores com funções de montagem integradas e que são capazes de produzir o código binário executável.

Embora um programa fonte possa ser traduzido diretamente para a linguagem objeto, o uso de uma representação intermediária independente de máquina tem as seguintes vantagens:

- Permitir o reaproveitamento de código, facilitando a portabilidade de um compilador para diversas plataformas; uma vez que apenas os módulos finais precisam ser refeitos a cada nova plataforma de *hardware*.

- A utilização de um otimizador de código que analise aspectos independentes de máquina e melhore o código intermediário antes de uma tradução definitiva.

Existem várias formas de representação de código intermediário, as duas mais comuns são: árvores de sintaxe e triplas de código intermediário (ou instruções de três endereços).

7.1 Árvores de Sintaxe

Uma árvore de sintaxe mostra a estrutura hierárquica de um programa fonte. Por exemplo, a atribuição a = b * c + b * -d poderia ser representada graficamente como é dado na figura 20 a seguir:

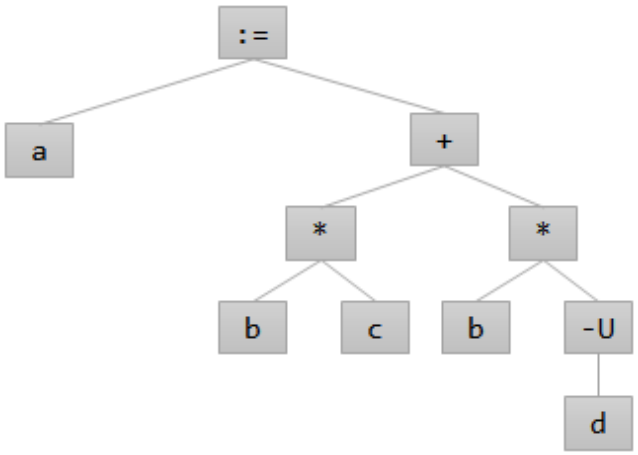


Figura 20 - Árvore sintática para a sentença a = b * c + b * -d

7.2 Triplas de Código Intermediário

As triplas de código intermediário são sequências de instruções dadas como código de três endereços, isto é, na forma "x := y op z" e que correspondem a estruturas básicas das linguagens de programação.

Nesse formato, os elementos "x", "y" e "z" podem ser variáveis ou constantes definidas pelo programador ou, ainda, variáveis temporárias geradas pelo compilador; o elemento "op" define o operador, isto é, caracteriza a operação a ser realizada (de natureza aritmética, relacional etc.); enquanto o ":=" define uma atribuição.

Para o comando de atribuição a = b * c + b * -d dado anteriormente como exemplo, obteríamos as seguintes triplas:

#tripla	operando-1	operador	operando-2	operando-3
1	t1	*	B	c
2	t2	-U	D	
3	t3	*	B	t2
4	t4	+	t1	t3
5	A	:=	t4	

Figura 21 - Exemplo de código intermediário para o comando a = b * c + b * -d

7.3 Tipos de Triplas

Bastante semelhantes às instruções em linguagem de montagem, podemos ter triplas com rótulos simbólicos e triplas para controle de fluxo. Veja alguns exemplos bastante comuns:

Natureza da operação	Exemplo de código	Equivalente em triplas	
Atribuição	x := y op z	1. 2.	t1 op y z x := t1
	x := -y	1. 2.	t1 -U y x := t1
	x := y	1. 2.	t1 := y x := t1
Desvio incondicional	goto L	1.	jmp L
Desvio condicional	if x oprel y then goto L	1.	¬ oprel x y L
Chamada a sub-rotina	sub(x_1, x_2, ..., x_n)	1. 2. ... n. n+1.	parm x_1 parm x_2 ... parm x_n call sub n
Ponteiros e endereços	x := &y	1. 2.	t1 address y x := t1
	x := *y	1. 2.	t1 l-ind y x := t1
	*x := y	1. 2.	t1 := y x s-ind t1
Índices em arrays	x := y[i]	1. ... m. ... ... n. p. p+1.	/* * avalia índice de y */ /* * calcula endereço de * y na posição i */ tp l-ind tn x := tp
	x[i] := y	1. ... m. ... ... n. p.	/* * avalia índice de x */ /* * calcula endereço de * x na posição i */ tn s-ind y

Figura 22 - As diferentes categorias de comandos da linguagem e suas correspondências na forma de triplas

O elemento ¬ *oprel* representa a operação lógica complementar descrita por *oprel*, enquanto os elementos *l-ind* e *s-ind* representam, respectivamente, as operações de *load-indirect*, que carrega o conteúdo de posição de memória dada por esse endereço; e *store-indirect*, que armazena na posição de memória dada por esse endereço.

7.4 A Geração de Código

Uma forma simples de geração de código intermediário pode ser construída a partir do mesmo mecanismo utilizado para a verificação de tipos, ou seja, uma gramática de atributos pode apresentar também regras que permitam a geração de código intermediário simultaneamente a ações semânticas de verificação de tipo.

Supondo uma linguagem hipotética qualquer, cuja gramática abaixo definisse a sintaxe para expressões aritméticas e comandos de atribuição:

$$
\begin{aligned}
&G = (\ \{A, E, T, F, E', T'\},\ \{id, +, *, (,)\},\ P,\\
&A)\\
&P = \{\quad A \rightarrow id := E \quad ,\\
&\qquad\quad E \rightarrow T\ E' \quad ,\\
&\qquad\quad E' \rightarrow + T\ E' \mid \varepsilon \quad ,\\
&\qquad\quad T \rightarrow F\ T' \quad ,\\
&\qquad\quad T' \rightarrow * F\ T' \mid \varepsilon \quad ,\\
&\qquad\quad F \rightarrow id \mid num\_int \mid num\_real \mid (\ E\ ) \quad \}
\end{aligned}
$$

Podemos acrescentar ações para a geração de código tais como:

Quando o analisador sintático aplicar as regras F → id ou F → num_int ou F → nro_real, podemos simplesmente empilhar em uma pilha de controle de operandos (PcO) o identificador, número inteiro ou número real correspondente (ou indicador de onde se encontra esse operando).

Quando o analisador sintático aplicar as regras E' → + T E' ou T' → * F T', sabemos que se trata de uma operação de adição ou multiplicação. Assim, se não houver problemas de compatibilidade entre os tipos, podemos gerar código intermediário utilizando os dois operandos que estiverem no topo da PcO e armazenando o resultado em uma variável temporária, que deverá ser deixada na PcO para operações futuras.

Reescrevendo as regras para introduzir ações de geração de código necessárias, teríamos:

A → id <G1> := E <G2>

```
<G1>  /* lembrar qual é o elemento do lado esquerdo de uma
      atribuição e iniciar o contador de variáveis temporárias
      em 1 */
      lado_esquerdo ← símbolo;
      prox_temp = 1;
<G2>  /* gerar efetivamente o código referente ao comando de
      atribuição */
      oper_1 ← pop( Pc0 );
      emitir( lado_esquerdo, ':=', oper_1 );
```

E' → + T E' <G3>

```
<G3>  /* geração do código para uma adição */
      oper_2 = pop( Pc0 );
      oper_1 = pop( Pc0 );
      result = make_temp();
      emitir( result, '+', oper_1, oper_2 );
```

T' → * F T' <G4>

```
<G4>  /* geração do código para uma multiplicação */
      oper_2 = pop( Pc0 );
      open_1 = pop( Pc0 );
      resul = make_temp();
      emitir( resul, '*', oper_1, oper_2 );
```

F → id <G5>

```
<G5>  /* encontrou um identificador como operador. Então,
      se o escopo for global, colocar seu nome na Pc0; se for
      local, colocar a referência relativa à pilha do programa
      (ex.: [base_da_pilha + endereço_relativo]) */
      pesquisa_tab_símbolo( símbolo );
      se símbolo.escopo = global
        push( Pc0, símbolo );
      Senão
        push( Pc0, [base_da_pilha+endereço_relativo] );
```

F → num_int <G6>

```
<G6>  /* encontrou um número inteiro como operador, colocar
      na Pc0 */
      push( Pc0, símbolo );
```

F → num_real <G7>

```
<G7>  /* encontrou um número real como operador, colocar na
      Pc0 */
      push( Pc0, símbolo );
```

7.5 Otimização

O uso de comandos elaborados, mais complexos, de forma aninhada ou encadeada, cujo uso é perfeitamente adequado em uma linguagem de alto nível, pode após a tradução não corresponder à melhor forma de codificação daquela tarefa em linguagem de baixo nível.

A tarefa de obter um código ótimo pertence à classe de problema NP, ou seja, além de não existir uma forma eficiente de solucioná-lo, o tempo necessário cresce exponencialmente de acordo com o tamanho do código, tornando inviável o processo. Na prática, o que os compiladores fazem é produzir um bom código, rápido e eficiente mesmo que essa versão não seja a melhor (ótimo).

Na maioria das vezes, o código gerado automaticamente pode ser ineficiente devido a redundâncias, instruções desnecessárias ou mesmo decorrentes de deficiências provenientes da codificação original. Entretanto, certas heurísticas podem ser aplicadas ao código em formato intermediário para otimizá-lo antes da produção do código em linguagem simbólica.

7.6 Eliminação de subexpressões comuns

É aplicável sempre que certas operações se repetem, mas sem que os seus argumentos sejam alterados. Por exemplo, considere o trecho de código a seguir em linguagem de alto nível:

```
x = a + b + c;
...
y = a + b + d;
```

A tradução desses comandos para uma linguagem de baixo nível, na forma de instruções de três endereços, produziria algo como:

```
_t1 := a + b
x := _t1 + c
...
_t2 := a + b
y := _t2 + d
```

Considerando que entre o primeiro comando e o segundo não houve qualquer atribuição às variáveis a e b, a operação de soma entre esses elementos poderia ser realizada uma única vez, o que resultaria no seguinte código:

```
_t1 := a + b
x := _t1 + c
...
y := _t1 + d
```

7.7 Propagação de cópias

Relativos a situações em que variáveis são utilizadas apenas para manter cópia de um valor e que, sem ter outros usos, poderiam ser eliminadas sem causar qualquer prejuízo.

7.8 Eliminação de código redundante ou sem efeito

Trata-se dos casos em que, por consequência do encadeamento dos conjuntos de instruções por exemplo, formam-se combinações em que as instruções se tornam duplicadas ou sem efeito.

Por exemplo, considere o caso de duas instruções idênticas de atribuição envolvendo variáveis. Se não houver outra atribuição a qualquer uma das variáveis entre as duas instruções, a segunda instrução de atribuição pode ser seguramente eliminada.

Também é possível que, por consequência de um arranjo de desvios, seja gerada uma instrução de desvio incondicional que direcione exatamente para a próxima instrução. Nesse caso, a instrução de desvio pode ser eliminada, preservando-se a sequência natural de execução.

7.9 Eliminação de código não alcançável

Embora bastante similar ao aspecto descrito no item anterior, as situações aqui envolvidas podem não ser tão óbvias. Trata-se da análise e da remoção de instruções que nunca serão executadas, pois não há um fluxo de execução que permita alcançá-las.

Considere o conjunto de instruções a seguir:

```
...
goto _L3
_t1 := x
_L4: _t2 := b + c
_L3: d := a + _t2
...
```

Note que o desvio incondicional dado pela instrução goto _L3 faz com que o fluxo de execução avance até a instrução com rótulo _L3, saltando as duas atribuições dadas entre elas. Por sua vez, qualquer desvio que direcione o fluxo de execução para o rótulo _L4 ou para um ponto qualquer anterior à instrução goto _L3, será incapaz de permitir a execução da atribuição à variável _t1, tornado essa instrução inalcançável.

7.10 Melhorias explorando propriedades algébricas

A própria decomposição de expressões maiores na forma de instruções de três endereços pode ocasionar em operações algébricas desnecessárias ou mesmo situações cujo resultado já seja conhecido, como é o caso de multiplicações por 1 ou somas com o valor zero.

Adicionalmente, certas operações apresentam propriedades associativas e/ou reflexivas que podem ser exploradas para obter um código mais enxuto ou que consumam menos ciclos de máquina para a sua realização. Por exemplo, as operações de multiplicação e divisão envolvendo números inteiros são realizadas como sucessivas operações de soma ou de subtração, seguidas de deslocamentos. Uma multiplicação ou divisão de um número inteiro por 2 pode ser realizada por meio de uma simples operação de deslocamento (shift), dispensando todo o processo geral usualmente empregado naquela operação. Outro exemplo é que ao invés de calcular o quadrado de um número, essa mesma operação pode ser realizada como uma multiplicação do valor por ele mesmo.

Exercícios resolvidos:

1. Considerando os tipos de triplas apresentados, qual seria o resultado da tradução para o código a seguir dado em linguagem de alto nível?

```
int main(void) {
   float A,B,C;
   float delta, x1, x2;
   printf("Entre com os coeficientes da equação\n");
   scanf("%f %f %f",&tA,&tB,&tC);
   delta = B*B - 4.*A*C;
     if (delta >= 0) {
     x1 = (-B + sqrt(delta))/(2.*A);
     x2 = (-B - sqrt(delta))/(2.*A);
   printf("As raízes reais são: 1a. raiz = %f 2a.raiz= %f\n", x1, x2);
}
else {
   printf("Não existem raízes reais");
}
return 0;
}
```

Resp.: Como as informações a respeito das variáveis estáticas e sub-rotinas estão armazenadas na tabela de símbolos, a alocação de memória e os endereços poderão ser determinados na geração final do código em linguagem de montagem. Aqui nos concentraremos no processo de tradução dos comandos para a forma de instruções de três endereços.

```
DATA:    A, B, C, delta, x1, x2
CODE:    param "Entre com os coeficientes da equação|n"
         _t1 := call printf 1
         param "%f %f %f"
         param A
         param B
         param C
         _t2 := call scanf 4
```

```
        _t3 := 4 * A
        _t4 := _t3 * C
        _t5 := B * B
        _t6 := _t5 - _t4
        delta := _t6
        lt delta 0 _L1
        _t7 := -U B
        param delta
        _t8 := call sqrt 1
        _t9 := _t7 + _t8
        _t10 := 2 * A
        _t11 := _t9 / _t10
        x1 := _t11
        _t12 := -U B
        param delta
        _t13 := call sqrt 1
        _t14 := _t12 - _t13
        _t15 := 2 * A
        _t16 := _t14 / _t15
        x2 := _t16
        param "As raízes reais são: 1a. raiz = %f 2a.raiz= %f\n"
        param x1
        param x2
        _t17 := call printf 3
        jmp _L2
_L1:    param "Não existem raízes reais"
        _t18 := call printf 1
_L2:     ret 0
```

2. Com base na gramática dada como exemplo, quais seriam a pilha de análise e o conjunto de triplas geradas para o comando: x=a+b*c?

Resp.: Procedemos a análise sintática da cadeia de entrada normalmente, ou seja, simulando a construção da árvore de derivação por meio de uma estrutura do tipo pilha pelas operações de Expansão pela Regra X e Verificação do Símbolo S. Adicionalmente, sempre que encontrarmos na pilha uma instrução para a geração de código, grafada por <Gn>, realizamos as tarefas especificadas naquela regra.

Pilha	Entrada	PcO	Obs.
$ A	x := a + b * c $		Expandir regra A → id := E
$ <G2> E := <G1> id	x := a + b * c $		Verifica(id): Ok
$ <G2> E := <G1>	:= a + b * c $		lado_esquer = x; prox_temp = 1;

$ <G2> E :=	:= a + b * c $		Verifica(:=): Ok
$ <G2> E	a + b * c $		Expandir regra E → T E'
$ <G2> E' T			Expandir regra T → F T'
$ <G2> E' T' F	a + b * c $		Expandir regra F → id
$ <G2> E' T' <G5> id	a + b * c $		Verifica(id): Ok
$ <G2> E' T' <G5>	+ b * c $	a	push(PcO, a)
$ <G2> E' T'	+ b * c $		Expandir regra T' → ε
$ <G2> E'	+ b * c $		Expandir regra E' → + T E'
$ <G2> <G3> E' T +	+ b * c $		Verifica(+): Ok
$ <G2> <G3> E' T	b * c $		Expandir regra T → F T'
$ <G2> <G3> E' T' F	b * c $		Expandir regra F → id
$ <G2> <G3> E' T' <G5> id	b * c $		Verifica(id): Ok
$ <G2> <G3> E' T' <G5>	* c $	a b	push(PcO, b)
$ <G2> <G3> E' T'	* c $		Expandir regra T' → * F T'
$ <G2> <G3> E' <G4> T' F *	* c $		Verifica(*): Ok
$ <G2> <G3> E' <G4> T' F	c $		Expandir regra F → id
$ <G2> <G3> E' <G4> T' <G5> id	c $		Verifica(id): Ok
$ <G2> <G3> E' <G4> T' <G5>	$	a b c	push(PcO, c)
$ <G2> <G3> E' <G4> T'	$		Expandir regra T' → ε
$ <G2> <G3> E' <G4>	$	a t1	t1 := b * c
$ <G2> <G3> E'	$		Expandir regra E' → ε
$ <G2> <G3>	$	t2	t2 := a + t1
$ <G2>	$		x := t2
$	$		Verifica($): Ok

Assim, a sequência de triplas (código intermediário) para a instrução seria:

*t1 := b * c*
t2 := a + t1
x := t2

3. Compare os modelos de máquina de pilha e máquina de registradores com relação às vantagens e às desvantagens que cada um oferece quando se trata de gerar código para avaliação de expressões.

Resp.: A geração de código para máquinas de registradores, geralmente, produz um código mais rápido e compacto do que a geração de código para máquina de pilha. No entanto, ela demanda um gerenciamento rigoroso sobre a alocação e o uso de cada registrador durante a compilação do programa-fonte, o que torna essa fase mais complexa do ponto de vista da implementação. O uso de registradores, no entanto, facilita a otimização de código dependente de máquina, como o reaproveitamento do conteúdo previamente carregado nos registradores. A geração de código para máquina de pilha é mais simples, mas produz código de qualidade inferior.

8 *ASSEMBLERS, LINKEDITORES* E CARREGADORES

Tanto os compiladores quanto montadores produzem um arquivo de binário contendo o código objeto associado ao arquivo-fonte de entrada. Observe que o arquivo com o código objeto contém parte da informação necessária à sua execução, mas para que ocorra a execução é preciso que esse código seja transferido para a memória principal. Caso o código ainda faça referências a elementos (dados ou rotinas) definidos externamente ao módulo, será preciso integrar essas referências ao código executável. Assim, as etapas envolvidas desde a tradução de um programa até a sua execução de um computador passam por uma série de módulos, que são:

Compilador: responsável pela tradução de um programa descrito em linguagem de alto nível para uma linguagem alvo, de baixo nível e próxima da linguagem de máquina (quando não a própria).

Montadores (*Assembler*): encarregados da tradução do programa escrito em linguagem *assembly*, resultando em um programa em linguagem de máquina.

Ligadores (*Linker* ou *Linkeditor*): responsáveis por unir diferentes partes de um programa e construir um único executável. Sua existência é fundamental para permitir a modularização de um programa e o uso de bibliotecas estáticas.

Carregadores (*Loader*): usualmente, partem do Sistema Operacional, é esse elemento que realiza a transferência de um programa do disco para a memória principal.

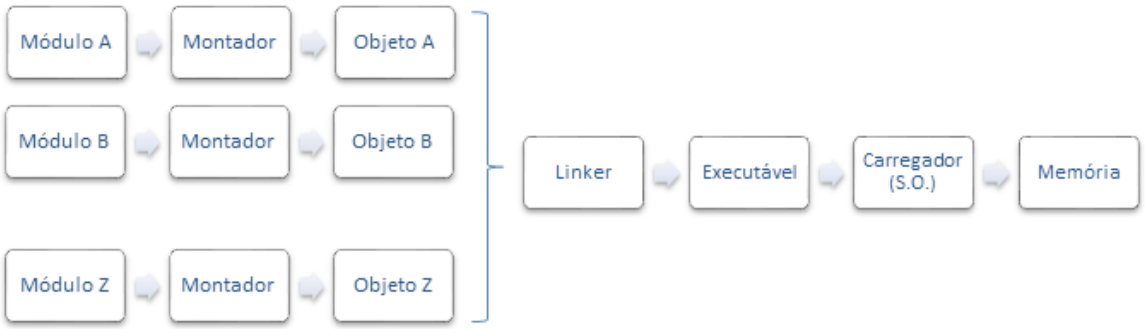
Vale ressaltar que o projeto de um compilador pode adotar diferentes arquiteturas, combinando e integrando em uma única ferramenta as funcionalidades de mais de um desses módulos.

Alguns exemplos de combinações desses elementos em um projeto poderiam ser:

Montador e carregador integrados: neste esquema tão logo a montagem do código fonte é feita, o resultado já em código de máquina absoluto é colocado diretamente na memória para a execução.

Montador e carregador independentes: quando esses dois elementos são separados, o processo de montagem do código fonte produzirá um código de máquina realocável, permitindo ao carregador colocá-lo em qualquer posição de memória que desejar.

Montador, ligador e carregador independentes: a separação da etapa de ligação das demais oferece mais flexibilidade ao processo, permitindo a compilação ou a montagem dos módulos do programa separadamente.



8.1 *Assemblers* (Montadores)

É de responsabilidade do montador a tradução dos programas escritos em linguagem de montagem (*assembly*) para linguagem de máquina, isto é, no conjunto de instruções da arquitetura.

A sintaxe típica de programas de montagem segue a forma:

[rótulo] [operação] [operando1] [, operando2] ; comentário

... em que: o *r*ótulo é um marcador empregado para definir um endereço da área de código; a **operação** pode ser tanto um *opcode* simbólico (mnemônico que representa uma instrução) quanto uma pseudoinstrução, que determina uma orientação para a atuação do montador; e os **operandos** são os elementos envolvidos na operação, usualmente registradores, constantes ou endereços de memória.

As funções de um montador compreendem quatro tarefas básicas:

1. A substituição dos *opcodes* mnemônicos pelas instruções com os *opcodes* numéricos do conjunto ISA (*Instruction Set of Architecture*), seguindo uma tabela de associações que relaciona o mnemônico com a instrução alvo.

2. A substituição dos endereços simbólicos que representam destinos de saltos e constantes por endereços numéricos; ou seja, determinar de maneira absoluta ou relativa em termos do valor do registrador PC (*Program Counter*) o endereço de destino dos rótulos.

3. Reservar espaço para dados de acordo com o tipo associado a cada variável declarada no programa.

4. Gerar constantes em memória para variáveis e constantes, determinando o valor associado ao modo de endereçamento do operando.

Para um melhor entendimento, suponhamos um processador hipotético baseado na arquitetura de Von-Neumann e que apresente o seguinte conjunto de instruções:

Opcode simbólico	Opcode numérico	Tamanho (em bytes)	Número de operandos	Ação
ADD	01	2	1	ACC ← ACC + mem(operando)
SUB	02	2	1	ACC ← ACC - mem(operando)
MUL	03	2	1	ACC ← ACC * mem(operando)
DIV	04	2	1	ACC ← ACC / mem(operando)
JMP	05	2	1	PC ← operando
JMPN	06	2	1	Se ACC < 0 então PC ← operando
JMPP	07	2	1	Se ACC > 0 então PC ← operando
JMPZ	08	2	1	Se ACC = 0 então PC ← operando
COPY	09	3	2	mem(operando2) ← mem(operando1)
LOAD	10	2	1	ACC ← mem(operando)
STORE	11	2	1	mem(operando) ← ACC
INPUT	12	2	1	mem(operando) ← entrada
OUTPUT	13	2	1	saída ← mem(operando)
STOP	14	1	0	Encerra a execução

Figura 23 - Conjunto de instruções (ISA) para uma arquitetura hipotética baseada no modelo de Von-Neumann

Assim, o programa escrito em linguagem de alto nível (dado abaixo, na coluna da esquerda) será traduzido pelo compilador para seu equivalente em linguagem de montagem (dado na coluna ao centro), enquanto o montador fará a tradução definitiva do código resolvendo os endereços e traduzindo os demais elementos.

Programa em linguagem de alto nível
int a,b,c;
read(a)
read(b)
c = a + b;
write(c);

Programa em linguagem de montagem (assembly)		
Rótulo	Mnemônico	Operando
	INPUT	N1
	INPUT	N2
	LOAD	N1
	ADD	N2
	STORE	N3
	OUTPUT	N3
	STOP	
N1:	SPACE	
N2:	SPACE	
N3:	SPACE	

Programa em linguagem de máquina		
Endereço	Opcode	Operando
00	12	13
02	12	14
04	10	13
06	01	14
08	11	15
10	13	15
12	14	
13	??	
14	??	
15	??	

Figura 24 - Exemplo de um programa de alto nível traduzido em Assembly e depois em linguagem de máquina

Os montadores podem realizar duas passagens no arquivo fonte. A primeira com o objetivo de reconhecer os símbolos definidos pelo programador (identificadores) e construindo a tabela de símbolos com seus respectivos valores, enquanto na segunda passagem é que efetivamente a geração do código objeto é realizada.

8.2 Formato do arquivo objeto

Tipicamente, um arquivo objeto é composto por cinco partes e contém os seguintes itens de informação:

Cabeçalho: contém a identificação do tipo de arquivo e dados sobre o tamanho do código e, eventualmente, o arquivo que deu origem ao arquivo objeto;

Código gerado: contém as instruções e os dados em formato binário, apropriado ao carregamento;

Relocação: contém as posições no código onde deverão ocorrer mudanças de conteúdo quando for definida a posição de carregamento; ou seja, a determinação dos endereços que serão utilizados no programa dados em função do endereço base de carregamento.

Tabela de símbolos: contém os símbolos globais definidos no módulo e símbolos cujas definições virão a partir de outros módulos;

Depuração: contém referências para o código fonte, tais como o número de linha, nomes originais dos símbolos locais e estruturas de dados definidas.

8.3 Carregador (*Loader*)

Se a arquitetura adotada define que a montagem e o carregamento são realizados por módulos separados, o montador produzirá um arquivo contendo o código de máquina e juntamente com as informações necessárias para que o carregador (*loader*) possa carregar o código de máquina para a memória e transferir a execução para o programa carregado.

Uma dessas formas é o chamado carregamento absoluto, em que os endereços são fixos e o programa é carregado sempre na mesma posição. Esse tipo de esquema é bastante limitado, porém permite uma compreensão de detalhes importantes da operação de carregadores e montadores.

A forma mais elementar para a execução do código de máquina gerado pelo montador é por meio do esquema *assemble and go*, no qual um único programa combina as tarefas associadas ao montador e do carregador. Essa estratégia combina as etapas de montagem e carregamento em um único programa e, portanto, não gera um arquivo com o módulo objeto.

Assim que o código de máquina é gerado pelo montador, é então colocado diretamente na posição de memória indicada pelo contador de localização. Ao término da montagem, o programa executável já estará na memória e o montador precisa simplesmente transferir o controle de execução para a primeira

instrução do código gerado.

O módulo de carregamento para o carregador absoluto é composto por dois tipos de registro:

Registro do tipo texto (ou tipo 0): contém informação que devem ser transferidas para a memória na posição indicada.

Registro do tipo fim (ou tipo 1): a informação do endereço para início da execução do programa deve ter apenas uma ocorrência no fim do módulo de carregamento

Suponha que um montador tenha gerado o código relativo ao conteúdo de dois seguimentos, sendo o primeiro equivalente a uma área de dados (variáveis do programa) e o segundo correspondendo à área de instruções, que contém o código de máquina associado à cada instrução do programa. A organização do módulo de carregamento segundo esse esquema seria composta por três registros:

0 00006000 4 00000000
0 00004000 E 30380000600031C0000060024E75
1 00004000

Os quatro campos do registro significam, respectivamente: tipo, endereço, tamanho (em bytes) e conteúdo do registro. Assim, a interpretação de cada registro deverá:

Como o campo tipo contém valor 0, isso indica que o registro é do tipo texto e, portanto, o seu conteúdo (dado no quarto campo), cujo tamanho equivale a quatro bytes (informado no terceiro campo) deverá ser transferido à memória a partir da posição $6000 (de acordo com o endereço que é dado no segundo campo).

De modo análogo, o segundo registro indica a transferência, a partir da posição $4000 de memória, dos 14 bytes cujo conteúdo é dado armazenado no quarto campo.

Por fim, o último registro está presente para a finalização do processo de carga, indicando que o controle de execução deverá ser transferido para a posição indicada no segundo campo (isto é, o endereço $4000).

Assim, um algoritmo simples como o que é dado a seguir poderia ser utilizado como carregador:

CarregadorAbsoluto(*módulo*)	
1	*arq* ← OpenFile(*módulo*)
2	**while** not EndOfFile(*arq*)
3	**do** *registro* ← ReadLine(*arq*)
4	*tipo* ← getTipoDoRegistro(*registro*)
5	*endereco* ← getEnderecoDoRegistro(*registro*)
6	**if** *tipo* == 0

```
7   then tamanho ← getTamanhoDoRegistro(registro)
8   código ← getConteudoDoRegistro(registro, tamanho)
9   MoveParaMemoria(endereço, código, tamanho)
10  else GoTo(endereço)
```

Figura 25 - Pseudocódigo de um carregador absoluto

8.4 Ligadores (*Linker*)

Tem por principal objetivo juntar diferentes arquivos objeto, gerando um único arquivo executável. Em outras palavras, o *linker* recebe como entrada os diversos módulos que devem ser conectados, gerando como saída um único módulo de carga. Na sequência, o programa carregador recebe o módulo de carga (executável) como entrada e o transfere seu código para a memória, ficando sob sua responsabilidade apenas os ajustes de relocação de acordo com o endereço base de memória.

É possível adotar estratégias combinadas, porém ao isolar os procedimentos de ligação e de carregamento por meio de programas separados, é possível uma economia na quantidade de memória alocada haja vista que o carregador compartilha a memória com o programa que está sendo executado.

Um *linker* deve ser capaz de resolver referências cruzadas, provenientes a partir de módulos montados de forma independente, bibliotecas de funções, fazendo uso de funções cujo código não é conhecido em todos os momentos do processo. Uma referência cruzada não resolvida é resultante de símbolos desconhecidos dentro de um módulo e que, normalmente, são definidos em outros módulos.

Durante o processo de montagem, o montador deve informar a existência de referências cruzadas e criar uma tabela de uso, inserindo-a juntamente no código objeto gerado. A tabela de uso tem por objetivo indicar quais símbolos externos foram utilizados pelo módulo e onde cada um deles foi referenciado.

Note que embora uma referência não resolvida possa ser entendida como um erro do ponto de vista do montador, na verdade, ela pode não ser, uma vez que sua definição pode ter sido feita em outro módulo interdependente. Uma forma comum para tratar essa questão é a inclusão de diretivas de montagem (por exemplo, a diretiva EXTERN na linguagem C). A desvantagem é que um erro de montagem dessa natureza somente é detectado durante o processo de ligação, quando todos os módulos são reunidos e há ainda alguma referência não resolvida.

8.5 Relocação e ligação

O esquema de montagem e carregamento absoluto apresenta uma forte limitação: o fato de que o programador deve ter acesso direto a posições de memória, especificando exatamente em que região da memória o programa e seus dados serão carregados.

A memória é um recurso controlado pelo sistema operacional e o programador não deve estar preso à necessidade de conhecer posições da memória física para que o seu programa funcione corretamente.

Por sua vez, desenvolver um programa completamente independente de sua localização é uma atividade complexa e, embora possível, caberá ao sistema resolver os problemas relacionados com posicionamento do código por meio do processo chamado de relocação.

Outro aspecto que também requer a colaboração do montador e do carregador para seu funcionamento é a combinação de módulos interdependentes, mas que tenham sido montados individualmente. Nesse contexto, é comum que um módulo faça referência a símbolos definidos em outros módulos e, assim, o montador recebe a informação de que um símbolo está definido em outro módulo ou de que um símbolo foi referenciado por outro módulo. Essa informação é então registrada junto ao módulo objeto para que seja utilizada pelo carregador, responsável pela resolução desses símbolos entre os módulos envolvidos.

Os dois tipos de ajustes que podem ocorrer no conteúdo do módulo objeto são:

- **Relocação:** ajuste interno ao segmento;
- **Ligação:** ajuste entre segmentos distintos.

A atividade de relocação é realizada conjuntamente por montadores e carregadores. Os montadores são encarregados de marcar as posições no código objeto passíveis de alteração, enquanto os carregadores devem reservar espaço suficiente na memória para receber o código de máquina e atualizar suas posições a partir da localização base do programa na memória. Assim, o módulo objeto deve conter também as informações adicionais que permitam realizar tais ajustes.

Outras informações que também deverão estar presentes no módulo objeto são aquelas que fazem referências aos símbolos externos. Nesse caso, há duas situações que podem ser tratadas e ambas devem estar presentes no módulo objeto:

- A primeira situação ocorre quando um símbolo é referenciado no segmento, mas ocorre em outro segmento e é usualmente descrita como uma referência externa (ER).
- A segunda é aquela em que um símbolo é definido nesse segmento e poderá ser referenciado em outro segmento, sendo descrito como uma definição local (LD) de um símbolo externamente referenciável.

Em um esquema que emprega carregadores de ligação direta, o montador deverá incluir no módulo objeto algumas estruturas de dados adicionais que incluam as informações descritas no parágrafo anterior:

Dicionário de Símbolos Externos (ESD): contém todos os símbolos que estão envolvidos no processo de resolução de referências entre segmentos: símbolos associados a referências externas (ER), definições locais (LD) ou definições de segmentos (SD);

Diretório de Relocação e Ligação (RLD): para cada segmento, indica quais posições deverão ter seus conteúdos atualizados de acordo com o posicionamento em memória deste e de outros segmentos.

A partir dessas estruturas, o carregador de ligação direta é capaz de definir os valores para todos os símbolos com referências entre segmentos e reajustar o conteúdo das posições afetadas pela relocação.

No exemplo que tratamos anteriormente, o montador oferecia como resultado um módulo objeto com dois tipos de registros: registro com código de máquina (tipo 0) e um registro de fim (tipo 1). Para o esquema de ligação direta, esse montador deveria ser modificado para fornecer outros dois tipos adicionais: um tipo para ESD e outro para RLD.

Os registros do tipo ESD conterão:

- Todos os símbolos definidos no segmento que podem ser referenciados por outros segmentos (lembrando que podem ser de dois tipos: definição do segmento ou definição local);
- Os símbolos que são referenciados, mas não definidos no segmento.

Assim, um registro desse tipo apresentará a seguinte estrutura:

Campo	Significado
1.	Tipo do registro (0),
2.	Símbolo
3.	Tipo de definição (SD – segmento, ou LD – local)
4.	Endereço relativo no segmento
5.	Campo de dimensão, com o comprimento em bytes

Nesse exemplo, as definições do tipo ER não receberão tratamento diferenciado e o campo de dimensão pode indicar tanto o espaço ocupado pelos dados de um símbolo (no caso de LD) como a dimensão total do segmento (no caso de SD).

Os registros do tipo TXT contêm o código de máquina e a informação do endereço relativo incorporada. Assim, o formato desse registro é:

Campo	Significado
1.	Tipo do registro (1)
2.	Endereço relativo
3.	Comprimento em bytes
4.	Código de máquina

Registros do tipo RLD indicam quais posições no segmento deverão ter conteúdo alterado de acordo com os endereços alocados aos segmentos e indicam também a partir de que símbolo o conteúdo deverá ser corrigido. Esses registros terão o seguinte formato:

Campo	Significado
1.	Tipo de registro (2)
2.	Posição relativa
3.	Comprimento em bytes
4.	Símbolo (base de ajuste)

Por fim, um registro do tipo END especificará o endereço de início de execução para o segmento que contém a "rotina principal", sendo dado na seguinte forma:

Campo	Significado
1.	Tipo de registro (3)
2.	Endereço de execução

Imagine que tenhamos um programa composto por dois módulos: o *main* e um *pgm*. O programa *main* obtém um valor inteiro entre 0 e 15 de uma variável chamada DIGIT e coloca na variável CHAR a representação equivalente em ASCII (isto é, um valor entre '0' e 'F'):

```
1    MAIN    MOVE.B DIGIT,D0
2            CMPI.B #10,D0
3            BLT ADD_0
4            ADDQ.B #('A'-'0'-10),D0
5    ADD_0   ADDI.B #'0',D0
6            MOVE.B D0,CHAR
7            RTS
8    CHAR    DS.W 1
9            END MAIN
```

Figura 26 - Código do módulo MAIN em linguagem de montagem

Observe que o código faz referência a um símbolo DIGIT que será definido externamente, pois a área de dados do programa define apenas a variável CHAR (linha 8).

No segmento em que DIGIT é efetivamente definido, torna-se necessário indicar que ele poderá ser referenciado externamente. Nesse exemplo, utilizamos a pseudoinstrução GLOB com esse objetivo. O código referente ao programa *pgm* é dado por:

```
1          GLOB DIGIT
2   PGM    MOVE.W VALUE,D0
3          MOVE.W D0,DIGIT
4          RTS
5   VALUE  DS.W 1
6   DIGIT  DS.W 1
7          END
```

Figura 27 - Código do módulo PGM em linguagem de montagem

A pseudoinstrução GLOB obrigará a criação de um registro do tipo ESD com tipo de definição LD e, assim, quando a posição relativa desse símbolo for determinada na tabela de símbolos locais, a informação do registro deverá ser complementada.

O montador deverá gerar os módulos objetos para cada uma das partes e o *linker* as combinará produzindo um único módulo. Como estamos tratando de um carregamento de ligação direta, o módulo único será colocado diretamente na memória, sem que seja criado em disco.

Após a montagem, suponha que o módulo objeto obtido para o segmento *main* seja:

```
0'MAIN'SD'00.1C
1.00.6.103900000000
1.06.4.0C00000A
1.0A.2.6D02
1.0C.2.5E00
1.0E.4.06000030
1.12.6.13C00000001A
1.18.2.4E75
1.1A.2.0000
2.02.4.'DIGIT'
2.14.4.'MAIN'
3.00
```

Figura 28 - Código de máquina gerado pelo montador para o módulo MAIN

Observando mais detalhadamente o módulo, encontramos: Diretório de símbolos externos – ESD (registros do tipo 0), código de máquina gerado – TXT (registros do tipo 1), diretório de relocação e ligação – RLD (registros do tipo 2) e registro de fim de segmento – END (registros do tipo 3), que deve especificar a posição relativa de execução, que é a posição 00.

De modo similar, o segmento *pgm* poderia ter o seguinte módulo:

```
0'.PGM".SD'.00.12
0'.DIGIT".LD'.10.2
1.00.6.30390000000E
1.06.6.33C000000010
1.0C.2.4E75
1.0E.2.0000
1.10.2.0000
2.02.4'.PGM'
2.08.4'.PGM'
3.
```

Figura 29 - Código de máquina gerado pelo montador para o módulo PGM

Apresentamos a seguir uma implementação possível para carregador de ligação direta, feita em três passos. A primeira etapa, dada pelo algoritmo a seguir, é responsável por alocar espaço contíguo de memória suficiente para todos os segmentos.

Para saber quanto espaço é necessário, a informação sobre o comprimento de cada segmento – presente em registros tipo ESD, com tipo de definição SD – é obtida. Os demais tipos de registro não são processados nesse passo. O resultado final desse algoritmo é a definição do endereço inicial do espaço de memória, indicado pela variável ipla, que receberá todos os segmentos.

	CarregadorLigadorDireto_etapa1(*listaDeMódulos*)
1	*total* ← 0
2	**for each** *módulo* **in** *listaDeMódulos*
3	**do** *arq* ← OpenFile(*módulo*)
4	*encontrado* ← **false**
5	repeat
6	*registro* ← ReadLine(*arq*)
7	*tipo* ← getTipoDoRegistro(*registro*)
8	**if** *tipo* == "ESD"
9	**then** *deftipo* ← getCampoDeDefinição(*registro*)
10	**if** *deftipo* == "SD"
11	**then** *tamanho* ← getTamanhoDoCampo(*registro*)
12	*total* ← *total* + 1
13	*encontrado* ← true
14	**until** *encontrado*
15	CloseFile(*arq*)
16	*ipla* ← AlocaMemoria(total)

Figura 30 - Pseudocódigo da primeira etapa do carregador e ligador direto

Uma vez determinado qual o endereço inicial de carregamento, IPLA (*Initial Program Load Address*), o carregador inicia a criação de uma Tabela de Símbolos Externos Globais – GEST. Para realizar essa tarefa, apenas as informações presentes em registros do tipo ESD (com tipos de definição SD e LD) são utilizadas. Assim, o segundo passo poderia ser implementado da seguinte forma:

	CarregadorLigadorDireto_etapa2(*listaDeMódulos, ipla*)
1	*GEST* ← CriarTabela()
2	*tamSeg* ← 0
3	*iniSeg* ← *ipla*
4	**for each** *módulo* **in** *listaDeMódulos*
5	**do** *arq* ← OpenFile(*módulo*)
6	**while not** EndOfFile(*arq*)
7	**do** *registro* ← ReadLine(*arq*)
8	*tipo* ← getTipoDoRegistro(*registro*)
9	**if** *tipo* == "ESD"
10	**then** *deftipo* ← getCampoDefinição(*registro*)
11	**if** *deftipo* == "SD"
12	**then** *valor* ← *iniSeg*
13	*tamSeg* ← getCampoTamanho(*registro*)
14	**else** *valor* ← *iniSeg* + getCampoPosicao(*registro*)
15	*símbolo* ← getCampoSimbolo(*registro*)
16	InsereNaTabela(*GEST, símbolo, valor*)
17	**else if** *tipo* == "END"
18	**then** *iniSeg* ← *iniSeg* + *tamSeg*
19	CloseFile(*arq*)

Figura 31 - Pseudocódigo da segunda etapa do carregador e ligador direto

Na fase de definição da GEST, um possível erro que poderia ser detectado é a duplicidade na definição de algum dos símbolos da tabela, refletindo situações em que um mesmo símbolo é redefinido em segmentos distintos.

No último passo do processo, o carregador irá realizar a transferência do código de máquina para a memória e transferir o controle da execução do programa para o endereço inicial do programa recém-carregado. O carregador voltará a tomar como endereço inicial de carregamento o valor ipla, lendo novamente cada módulo objeto na sua sequência original.

A variável pontoDeInício registrará a posição de início de execução para o segmento que definir um registro do tipo END com argumento.

Quando um registro ESD for lido, o único processamento necessário será a obtenção do comprimento do segmento, permitindo a atualização correta da variável que indica a posição inicial de carga de cada segmento, iniSeg. Essa informação está no registro ESD com tipo de definição igual a SD (*Segment Definition*).

Os registros do tipo TXT deverão ter seu conteúdo transferido para a memória principal. O conteúdo de cada campo do registro (posição relativa ao início do segmento, tamanho e conteúdo) é obtido e o endereço de destino é resolvido tomando por base o valor de iniSeg. Ao final da transferência, as posições indicadas em registros do tipo RLD deverão ter seu conteúdo ajustado a partir da informação registrada na GEST.

Os registros do tipo RLD devem ter suas indicações de posição relativa corrigida, sendo que o endereço de memória cujo conteúdo será alterado, endereço, é obtido a partir da combinação dessa informação e do endereço de início do segmento, iniSeg. O novo valor é especificado pelo campo de símbolo presente nesse registro (o símbolo é lido do registro e seu valor é obtido a partir de uma busca na GEST). Nesse ponto, é possível detectar erros associados a símbolos que tenham sido referenciados, mas não tenham definido em nenhum módulo.

Assim, a terceira etapa do carregamento direto poderia ser dada pelo seguinte algoritmo:

```
CarregadorLigadorDireto_etapa3(listaDeMódulos, ipla, GEST)
1   pontoDeInício ← ipla
2   iniSeg ← ipla
3   tamSeg ← 0
4   for each módulo in listaDeMódulos
5   do arq ← OpenFile(módulo)
6    while not EndOfFile(arq)
7    do registro ← ReadLine(arq)
8    tipo ← getTipoDoRegistro(registro)
9    switch tipo
10   case "ESD":
11   deftipo ← getCampoDefinição(registro)
12   if deftipo == "SD"
13   then tamSeg ← getCampoTamanho(registro)
14   case "TXT":
15   endereço ← iniSeg + getCampoPosicao(registro)
16   tamanho ← getCampoTamanho(registro)
17   código ← getCampoConteudo(registro)
18   MoveParaMemoria(endereço, código, tamanho)
19   case "RDL":
20   endereço ← iniSeg + getCampoPosicao(registro)
21   tamanho ← getCampoTamanho(registro)
22   símbolo ← getCampoSimbolo(registro)
23   base ← BuscaNaTabela(GEST, símbolo)
24   MoveParaMemoria(endereço, valorAntigo, tamanho)
25   novoValor ← valorAntigo + base
26   MoveParaMemoria(endereço, novoValor, tamanho)
```

```
27  case "END":
28  endereço ← getCampoPosicao(registro)
29  if endereço ≠ nulo
30  then pontoDeInício ← iniSeg + tamSeg
31  iniSeg ← iniSeg + tamSeg
32  CloseFile(arq)
33  GoTo(pontoDeInício)
```

Figura 32 - Pseudocódigo da terceira etapa do carregador e ligador direto

Leituras recomendadas:

Capítulo 7 do livro "Compiladores: Princípio, Técnicas e Ferramentas" (livro do Dragão)

Capítulos 7 e 8 do livro "Implementação de Linguagens de Programação: Compiladores"

Exercícios resolvidos:

1. Descreva a estrutura de um *frame* típico utilizado no ambiente de execução de uma linguagem de programação estruturada em blocos. Explique o que é e para que serve cada um de seus campos.

Resp.: O frame comporta as variáveis locais de cada bloco (acessadas por meio de offsets positivos em relação ao LB) e também as seguintes informações:

Link dinâmico: serve para armazenar o LB do frame anterior;

Link estático: serve para armazenar o LB do frame mais recentemente ativado e que envolve, estaticamente, o bloco correspondente ao frame corrente. Serve para acessar variáveis não locais.

Endereço de retorno: serve para armazenar o endereço da instrução que deve ser executada no retorno no bloco correspondente ao frame corrente.

Caso o bloco corrente seja um procedimento ou função parametrizado, os parâmetros são acessados por meio de off-set negativo em relação ao LB.

A base do frame é referenciada por um ponteiro fixo durante toda a execução do bloco, denominado LB. O topo do frame é apontado por um registrador denominado SB, que flutua durante a avaliação de expressões.

2. Suponha que um endereço ocupe dois bytes, um integre também dois bytes, o tipo boolean igual a um byte e o tipo real a quatro bytes. Considere a seguinte estrutura de um programa Pascal:

```
program P;
var A,B: integer;
function Q (N: boolean): real;
   var E,F,G: boolean;
   begin
   ...
   end;
procedure R (M: real);
   var C,D: real;
   procedure S ():
     var E: real;
     begin
     ...
     end;
   begin
   ...
   end;
begin
...
end.
```

a) Considere o fluxo de execução P → R → S e determine o endereço de cada uma das variáveis e parâmetros visíveis (deslocamento + registrador) nessa situação;

Resp.: Os endereços são dados em relação a base do frame, indicado pelo ponteiro LB. Assim, teríamos E: 6[LB], C: 6[L1], D: 10[L1], M: -4[L1], A: 0[SB], B: 2[SB]

b) Considere o fluxo de execução P → R → S e determine a configuração da pilha de execução nessa situação.

Resp.: *Posição* *Conteúdo*

Endereço base: A(2)
B(2)
*** Frame R ***
M(4)
LD (vazio ou SB)
LE (vazio ou SB)
ER
C(4)
D(4)
*** Frame S ***

LD(R)
LE(R)
ER
Topo: E(4)

c) Considere o fluxo de execução P → R → S → S → Q e determine o endereço de cada uma das variáveis e parâmetros visíveis (deslocamento + registrador) nessa situação;

Resp.: Nesse caso, teríamos E: 6[LB], F: 7[LB], G: 8[LB], N: -1[LB], A: 0[SB] e B: 2[SB]

d) Considere o fluxo de execução P → R → S → S → Q e determine a configuração da pilha de execução nessa situação.

Resp.: Posição Conteúdo

Endereço base: A(2)
B(2)
*** Frame R **M(4)*
LD (vazio ou SB)
LE (vazio ou SB)
ER
C(4)
D(4)
*** Frame S1 ***
LD(R)
LE(R)
ER
E(4)
*** Frame (S2) ***
LD(S1)
LE(R)
ER
E(4)
*** Frame (Q) ***
N(1)
LD(S2)
LE(vazio ou SB)
ER
E(1)
F(1)
Topo: G(1)

Interativa

Informações:
www.sepi.unip.br ou 0800 010 9000