Implementação completa de um compilador

Material baseado no cap. 5 do livro

“Introdução à Compilação”

João José Neto

Introdução

Uma vez estudados diversos aspectos dos fundamentos pertinentes ao estudo dos compiladores, e levantadas as especificações relacionadas à sua implementação, a melhor forma de fixar todos os conceitos é aplicar na prática os métodos e técnicas estudados, com a finalidade de obter um compilador para uma linguagem desejada.

Para tanto, antes de mais nada é necessário que sejam efetuadas diversas decisões de projeto, especialmente em relação aos itens para os quais tenham sido levantadas múltiplas opções.

Uma vez escolhido o roteiro de implementação, é relativamente simples compor o programa final aplicando cuidadosamente os métodos de construção escolhidos para cada parte do compilador.

Integrar as partes do compilador também não é uma tarefa complexa, pois existem muitos algoritmos suficientemente conhecidos e divulgados na literatura que implementam as técnicas escolhidas.

Neste capítulo, é desenvolvido um exemplo completo, procurando-se através dele fornecer ao leitor um roteiro que possa servir como base para a elaboração de projetos semelhantes.

Especificações Gerais do Projeto

A primeira importante tarefa do projetista consiste em determinar alguns vínculos de projeto que deverão nortear todo o desenvolvimento a ser efetuado.

Entre os principais pontos a considerar destacam-se:

(a) finalidade a que se destina o compilador.
(b) requisitos de velocidade do compilador.
(c) requisitos de velocidade do código-objeto.
(d) máquina hospedeira do compilador.
(e) máquina-alvo a ser utilizada.
(f) sistemas operacionais disponíveis em ambas as máquinas.
(g) ambiente de execução oferecido pela máquina-alvo.
(h) grau de complexidade da linguagem-fonte.
(i) número desejado de passos de compilação.
(j) requisitos de compatibilidade, no nível do programa-objeto.
(k) recursos computacionais disponíveis.
(l) técnicas a serem adotadas para a implementação do compilador.

Por questão de simplicidade, a finalidade prevista para o compilador no exemplo desenvolvido a seguir é a de servir apenas como instrumento didático.

Dispensam-se as características de eficiência, estabelecendo-se a necessidade de um compilador simples e compacto, cujo tempo de desenvolvimento seja tão pequeno quanto possível.

O compilador deverá ser transportável, razão pela qual seu desenvolvimento deverá ser feito em uma linguagem de alto nível de uso amplamente difundido.

Neste trabalho, optou-se pelo uso de pseudo-código, portanto não foi utilizada nenhuma linguagem de programação específica, estruturando-se as ideias de forma tal que possam ser implementadas em qualquer linguagem de programação algorítmica, bastando para isso um pequeno esforço de codificação.

Como se trata de um compilador com finalidade didática, é possível selecdionar, para a construção do código-objeto, um subconjunto muito restrito de instruções de máquina, efetuando-se a geração do código em uma linguagem simbólica (do tipo "assembly").

Com essa escolha, simplifica-se o compilador repassando totalmente ao montador os problemas de gerenciamento de memória, para as instruções e para os dados.

Mantidas tais referências na forma simbólica, repassa-se ao montador a tarefa de gerenciá-los, efetuando alocações de memória, resolução de referências à frente, preenchimento de endereços dos operandos em instruções de referência à memória, etc.

Neste exemplo, não serão exigidos grandes recursos do sistema operacional hospedeiro, bastando que estejam disponíveis funções ou sub-rotinas de biblioteca do sistema que se encarreguem de efetuar operações de entrada e saída.

Para maior simplicidade na operação de geração de código, pode-se impor que tais recursos ofereçam entrada e saída formatados, exigência essa que na maioria dos sistemas em uso é facilmente atendida.

Quanto ao ambiente de execução, exploram-se apenas os recursos mencionados da linguagem de máquina e do sistema operacional, de modo que o ambiente de execução resulte o mais simples possível.

Desta maneira, ficam automaticamente preenchidos os requisitos de compatibilidade do programa-objeto com o sistema operacional e com as bibliotecas do ambiente de execução utilizadas para outras finalidades.

Com as especificações descritas, não são exigidos quaisquer recursos computacionais especiais, ficando o projeto totalmente compatível para que seja implementado em qualquer sistema computacional disponível.

Em relação à maneira de implementar o compilador, pode-se optar pela construção de um simulador dos autômatos a serem utilizados, dirigido por tabelas que especificam as transições a serem executadas em cada caso.

Um mesmo interpretador de tabelas de transições pode ser utilizado para simular tanto os autômatos finitos como as submáquinas de autômatos de pilha estruturados.

Para simplificar as tabelas, opta-se pelo uso de ações semânticas especiais para promover "look-ahead", chamadas e retornos de submáquinas.

Especificação da linguagem-fonte

- **Para viabilizar a implementação dos reconhecedores necessários à construção do esqueleto de um compilador dirigido por sintaxe, é indispensável que, antes de mais nada, a linguagem a compilar seja descrita formalmente através de uma gramática livre de contexto ou regular que a aproxime adequadamente.**
- **Caso se trate de alguma linguagem já formalizada, convém obter um relatório de especificação da mesma, bem como outros documentos nos quais estejam descritas suas normas de padronização.**
- **A importância dessa atividade é muito grande, uma vez que deste cuidado depende a futura compatibilidade do compilador a ser construído com outros softwares relacionados a essa linguagem.**
- **Caso se trate de uma linguagem nova, deve existir um trabalho prévio de projeto e especificação formal, que levem à construção de uma gramática em que o projetista deverá se basear para a obtenção do compilador.**
- **Neste texto não são cobertos os aspectos do projeto de linguagens, uma vez que este assunto não pertence ao seu escopo, podendo-se encontrar muita informação neste sentido na literatura específica sobre características de linguagens de programação e seu projeto.**

Embora desnecessário na maioria dos casos, para o propósito deste exemplo, é muito desejável que a linguagem esteja formalizada através da notação de Wirth ou de diagramas de sintaxe, os quais tornam mais direta e intuitiva a construção do reconhecedor segundo o método aqui estudado anteriormente.

De qualquer forma, através da aplicação das regras de conversão de notação das gramáticas (Seção 3.1), é sempre possível convertê-la para a notação de Wirth, para possibilitar a aplicação do método de construção de reconhecedores descrito em 3.3.

Desejando-se, entretanto, aplicar as técnicas apresentadas em 3.2, pode-se obter com uma certa facilidade bons reconhecedores descendentes recursivos, LL(1) ou ascendentes LR(1), através de manipulações convenientes da gramática original.

Para o exemplo a ser desenvolvido neste capítulo, foi escolhida uma linguagem muito simples, definida a seguir na notação BNF.

O formato em que a linguagem deve ser escrita é livre: espaços entre os símbolos que representam átomos são ignorados, servindo apenas como separadores para efeito de visualização do texto-fonte.

Entre quaisquer dois átomos pode ser inserido um comentário, que é iniciado com o símbolo %, e que se estende até o final da linha em que ocorre.

O final de linha também funciona como separador entre dois átomos.

Separadores só são obrigatórios em caso de ambigüidade.

Descrição informal da linguagem

- Declarações
- Comandos imperativos
- Comandos de controle

Formalização da linguagem

- Sintaxe da linguagem em Notação BNF
- Sintaxe da linguagem em Notação de Wirth
- Sintaxe da linguagem na forma de Autômato de Pilha Estruturado

Descrição BNF da Linguagem-Exemplo

```
(1) <programa> ::= <seqüência de comandos> END
(2) <seqüência de comandos> ::= <comando> | <seqüência de comandos>; <comando>
(3) <comando>::= <rótulo>: <comando> | <atribuição> | <desvio> | <leitura>
                 | <impressão> | <decisão> | ε
(4) <atribuição> ::= LET <identificador> := <expressão>
(5) <expressão> ::= <expressão> + <termo>  | <expressão> - <termo> | <termo>
(6) <termo>::= <termo> * <fator>  | <termo> / <fator>  | <fator>
(7) <fator> ::= <identificador> | <número> | < <expressão> >
(8) <desvio> ::= GO TO <rótulo> | GO TO <identificador> OF <iista de rótulos>
(9) <lista de rótulos> ::= <rótulo> | <lista de rótulos> , <rótulo>
(10) <rótulo> ::= <identificador>
(11) <leitura> ::= READ <lista de identificadores>
(12) <lista de identificadores>::=<identificador>,<lista de identificadores>|ε
(13) <impressão> ::= PRINT <lista de expressões>
(14) <lista de expressões> ::= ε |  <expressão>, <lista de expressões>
(15) <decisão> ::= IF <comparação> THEN <comando> ELSE <comando>
(16) <comparação> ::= <expressão> <operador de comparação> <expressão>
(17) <operador de comparação>:: =   >  |  =  |  <
(18) <identificador>::=<letra>|<identificador><letra>|<identificador><dígito>
(19) <número> ::= <dígito> | <número> <dígito>
(20) <letra> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
(21) <dígito> ::= 0|1|2|3|4|5|6|7|8|9
```

Analisador Léxico

- Autômato reconhecedor da linguagem léxica
- Ações complementares
- Exemplo de operação do analisador léxico
- A.L. como Passo Independente de Compilação
- A.L. como Função do Reconhecedor Sintático

Reconhecedor Sintático

- Integração com o Analisador Léxico
- Obtenção do Autômato de Pilha Estruturado
- Eliminação de Algumas Redundâncias
- Exemplo de Operação do Reconhecedor

Sintaxe da Linguagem, em Notação de Wirth iterativa

```
(1) programa = seqüência-de-comandos "END".
(2) seqüência-de-comandos = comando { ";" comando }.
(3) comando ={ rótulo ":" }[ atribuição | desvio | leitura | impressão | decisão ].
(4) atribuição = "LET" identificador ":=" expressão.
(5) expressão = termo { ( "+" | "-" ) termo }.
(6) termo = fator { ( "*" | "/" ) fator }.
(7) fator = identificador | número | "(" expressão ")".
(8) desvio = "GO" "TO" ( rótulo | identificador "OF" lista-de-rótulos ).
(9) lista-de-rótulos = rótulo { "," rótulo }.
(10) rótulo = identificador.
(11) leitura = "READ" lista-de-identificadores.
(12) lista-de-identificadores = [ identificador { "," identificador } ].
(13) impressão = "PRINT" lista-de-expressões.
(14) lista-de-expressões = [ expressão { "," lista-de-expressões } ].
(15) decisão = "IF" comparação "THEN" comando "ELSE" comando.
(16) comparação = expressão operador-de-comparação expressão.
(17) operador-de-comparação = ">" | "=" | "<".
```

Gramática Léxica, em Notação de Wirth

```
Tokens não triviais:
(18) identificador = letra { letra | dígito }.
(19) número = dígito { dígito }.
(20) letra = A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z.
(21) dígito = 0|1|2|3|4|5|6|7|8|9.
```

```
Palavras reservadas:
"END", "LET", "GO", "TO", "OF", "READ", "PRINT", "IF", "THEN", "ELSE"
```

```
Operadores e outros sinais:
";", ":", "+", "-", "*", "/", "(", ")", ",", ">", "=", "<"
```

```
Símbolos compostos:
":="
```

Descrição informal dos comandos

Programa

Um programa qualquer, denotado na linguagem desejada, deve ser formado através da construção de uma seqüência de comandos, separados por ponto e vírgula, sendo o comando vazio uma alternativa válida.

Os rótulos que precedem um comando dão nome ao mesmo para que possam ser referenciados em comandos de desvio. Cada comando pode receber mais de um nome.

Ao longo de todo o programa, cada rótulo só pode aparecer uma única vez como nome de algum comando.

Não há restrições quanto ao número de instâncias dos rótulos-destino dos desvios permitidos na linguagem.

Além do comando vazio, cinco outros estão disponíveis: o comando de atribuição de valor a uma variável, o comando de desvio incondicional, o comando de leitura de valores, o comando de impressão de valores, e o comando de decisão.

Comando vazio

- O **comando vazio** não tem função alguma no programa, podendo ser livremente utilizado como mero recurso lingüístico de programação.

Comando de atribuição

- O comando de **atribuição de valor a uma variável** envolve o cálculo do valor associado a uma expressão, valor esse que é depositado em uma variável, explicitada através do seu identificador à esquerda da expressão.

Declaração de variáveis

- A linguagem não possui declarações explícitas para variáveis, sendo estas consideradas como declaradas na primeira ocasião em que forem referenciadas.
- Cabe portanto ao programador cuidar para que suas variáveis sejam adequadamente utilizadas ao longo do programa.

Expressões

- A avaliação de **expressões** segue as convenções usuais, sendo efetuada da esquerda para a direita, tendo produtos e divisões precedência sobre somas e subtrações, precedência essa que pode ser alterada através da utilização de parênteses.
- Apenas números e identificadores de variáveis são aceitos em expressões.
- Sobre os valores que representam, incidem as quatro operações aritméticas habituais.
- Essa linguagem utiliza aritmética inteira, com a precisão que esteja disponível na máquina na qual o programa-objeto deve ser executado.

Comandos de desvio

- Os **comandos de desvio** apresentam-se em dois formatos:
  - no primeiro (desvio direto), é especificado um desvio incondicional para um rótulo explicitamente indicado;
  - no segundo (desvio múltiplo), um identificador, que representa uma variável inteira, é utilizado como índice para a seleção de um dos rótulos de uma lista, e é para o rótulo assim selecionado que se executa o desvio. O primeiro rótulo desta lista associa-se ao valor zero da variável indexadora, o segundo, ao valor um, e assim por diante. Valores indevidos para a variável indexadora tornam inócuo o comando.

Comando de leitura

- O comando de **leitura** promove a entrada de dados de um meio externo (por exemplo, o teclado) e, com os valores lidos, preenche as variáveis de uma lista especificada no comando.
- No momento da execução deste comando, cada um dos dados deve ser fornecido como um elemento separado (por exemplo, um dado por linha).
- Assim, para um comando que especifica a leitura de n variáveis, deverá ser promovida a leitura de n linhas de dados, cada qual referente a uma das variáveis, e fornecidos na ordem especificada pela lista de variáveis especificada no comando.
- Caso essa lista seja vazia, o comando de leitura ignora (salta) a próxima linha de dados fornecida.

Comando de impressão

- O comando de **impressão** opera de forma análoga, porém age no sentido inverso: cada uma das expressões da lista que consta no comando é avaliada, e os valores obtidos são impressos, um por linha, na mesma ordem em que as expressões correspondentes figuram na lista especificada.
- Caso a lista encontrada no comando seja vazia, apenas uma linha em branco é impressa.

Comparações

- Comparações entre expressões são efetuadas através dos operadores > (maior que), = (igual a) e < (menor que).
- As expressões são avaliadas e comparadas da maneira habitual, de acordo com o operador escolhido, sendo todos os valores tratados como números inteiros relativos.
- Operações de comparação são usadas em comandos de decisão.

Comando de decisão

- O **comando de decisão** efetua inicialmente uma comparação entre duas expressões.
- Caso a comparação resulte verdadeira, será executado o comando que estiver presente, no comando de decisão, entre as palavras reservadas THEN e ELSE.
- Caso contrário, executa-se o comando que figurar após a palavra reservada ELSE.
- Note-se que ambos esses comandos são obrigatórios em todos os comandos de decisão desta linguagem.

Observações

- O compilador deverá aceitar todos os programas sintaticamente corretos, convertê-los para o formato de programa-objeto e prepará-los para a execução.
- Não está indicada, na especificação desta linguagem, a detecção de nenhum erro de execução, ficando portanto a inclusão desta função a critério de quem elaborar o compilador.

Casos anormais

Casos anormais deverão sofrer um tratamento pré-determinado:

- **em expressões que causem "overflow", esta ocorrência é ignorada.**
- **variáveis não preenchidas pelo programa são associadas ao valor que estiver contido na posição de memória na qual tenham sido alocadas.**
- **uma operação de divisão por zero não é executada, e nesse caso o próprio dividendo será considerado como resultado.**
- **em uma operação de divisão cujo resultado seja fracionário, apenas a parte inteira do quociente é considerada como sendo o resultado.**
- **nas operações de multiplicação, os resultados ocupam o dobro dos bits ocupados pelos fatores, mas nesta implementação apenas a palavra menos significativa resultante dessa operação é considerada como sendo o resultado.**
- **desvios indexados com índice inválido são ignorados, resultando um comando inócuo.**
- **na leitura, são considerados apenas os bits correspondentes à palavra menos significativa do dado fornecido, caso este tenha amplitude muito grande.**

Especificação da linguagem-objeto

Linguagem da máquina virtual

utilizada no livro

Introdução à Compilação

Especificação da linguagem de saída

Para determinar o conjunto de instruções da máquina-alvo a ser empregado pelo compilador, optou-se por reduzir a um pequeno número as instruções utilizadas, de tal modo que o subconjunto assim escolhido possa ser facilmente adaptado para alguma arquitetura existente, a ser utilizada em uma implementação.

Escolhe-se ainda um conjunto muito reduzido de pseudo-instruções, composto de pseudo-instruções típicas dos montadores usualmente disponíveis.

Para uma implementação real, eventuais discrepâncias entre o conjunto de códigos aqui proposto e o disponível na máquina podem ser contornados com facilidade através de adaptações triviais.

As instruções de máquina a serem utilizadas são todas instruções de referência à memória, com endereçamento direto ou imediato, e estão relacionadas na tabela seguinte:

Linguagem da máquina virtual

Conjunto minimalista de instruções da máquina-alvo

Instrução		Interpretação
LDA	X	Obtém o conteúdo da posição X, e deposita no acumulador.
STA	X	Armazena o conteúdo do acumulador na posição X.
ADA	X	Soma ao acumulador o conteúdo da posição X.
SUB	X	Subtrai, no acumulador, o conteúdo da posição X.
MUL	X	Multiplica, no acumulador, o conteúdo da posição X.
DIV	X	Divide o acumulador pelo conteúdo da posição X.
CALL	X	Chama a subrotina X (da biblioteca de execução).
BRU	X	Desvia incondicionalmente para a posição X.
BRN	X	Desvia para a posição X se o conteúdo do acumulador for negativo.
BRZ	X	Desvia para a posição X se o conteúdo do acumulador for zero.
BRP	X	Desvia para a posição X se o conteúdo do acumulador for positivo.

Pseudo-instruções

Conjunto de pseudo-instruções do montador

Pseudo-instrução	Interpretação
LBL X	**Atribui ao endereço da próxima instrução o rótulo X.**
DS X	**Define posição de memória com conteúdo inicial igual ao número X.**
END	**Demarca o final físico do programa-fonte.**
EXT X	**Define o rótulo X como nome de rotina externa, chamada pelo código-objeto (rotina de biblioteca).**

Formatação do código simbólico

- Cada linha do código simbólico pode conter uma ou mais instruções e/ou pseudo-instruções separados entre si por um ponto e vírgula, ordenadas da esquerda para a direita, e as linhas são ordenadas de cima para baixo.
- Comentários podem ser inseridos na forma de um texto, introduzido por %, que se estende até o final da linha.
- Operandos referentes a posições de memória são representados por identificadores.
- Identificadores representados por # seguido de um inteiro simbolizam endereços de memória conhecidas apenas pelo compilador.
- Operandos imediatos representam valores e não endereços, e são denotados por = seguido de um inteiro.
- Como no texto-fonte, inteiros são denotados em decimal

Linguagem da MVN

utilizada no Laboratório de Fundamentos e no projeto da disciplina

Mnemônicos da ling. máquina MVN

Operação **0** **Jump** Mnemônico **JP**	Operação **1** Jump if **Zero** Mnemônico **JZ**	Operação **2** Jump if **Negative** Mnemônico **JN**	Operação **3** Load **Value** Mnemônico **LV**
Operação **4** **Add** Mnemônico **+**	Operação **5** **Subtract** Mnemônico **–**	Operação **6** **Multiply** Mnemônico *****	Operação **7** **Divide** Mnemônico **/**
Operação **8** **Load** Mnemônico **LD**	Operação **9** Move to **Memory** Mnemônico **MD**	Operação **A** **Subroutine** Call Mnemônico **SC**	Operação **B** **Return** from Sub. Mnemônico **RS**
Operação **C** **Halt** Machine Mnemônico **HM**	Operação **D** **Get** Data Mnemônico **GD**	Operação **E** **Put** Data Mnemônico **PD**	Operação **F** **Operating** System Mnemônico **OS**

Mnemônicos disponíveis para a MVN

```
; MNEMÔNICOS    CÓDIGO    INSTRUÇÃO / PSEUDO-INSTRUÇÃO
;
;    JP    J      /0xxx     JUMP INCONDICIONAL
;    JZ    Z      /1xxx     JUMP IF ZERO
;    JN    N      /2xxx     JUMP IF NEGATIVE
;    LV    V      /3xxx     LOAD VALUE
;    +     +      /4xxx     ADD
;    -     -      /5xxx     SUBTRACT
;    *     *      /6xxx     MULTIPLY
;    /     /      /7xxx     DIVIDE
;    LD    L      /8xxx     LOAD FROM MEMORY
;    MM    M      /9xxx     MOVE TO MEMORY
;    SC    S      /Axxx     SUBROUTINE CALL
;    RS    R      /Bxxx     RETURN FROM SUBROUTINE
;    HM    H      /Cxxx     HALT MACHINE
;    GD    G      /Dxxx     GET DATA
;    PD    P      /Exxx     PUT DATA
;    OS    O      /Fxxx     OPERATING SYSTEM CALL
;
;    @     @                ORIGIN
;    #     #                END
;    K     K                CONSTANT
```

Outras possíveis máquinas virtuais

Que podem ser utilizadas no desenvolvimento de projetos desta natureza

Outras máquinas virtuais

- Há muitas máquinas virtuais em uso, que podem ser utilizadas no lugar das mencionadas anteriormente, entre as quais duas podem ser destacadas:
  - **P-machine** – a linguagem intermediária ***P-code***, que Wirth utiliza em sua implementação portátil da linguagem Pascal, é interpretada pela máquina virtual P-machine, e constitui uma possível opção como linguagem de saída de compiladores para outras linguagens de programação.
  - **Java Virtual Machine** – a linguagem intermediária ***Java Bytecode***, interpretada pela máquina virtual Java Virtual Machine, aderente às diversas características da linguagem Java, pode também ser usada como linguagem de saída em compiladores de outras linguagens de programação de alto nível.

Critérios p/ a geração de código (1)

O texto-fonte é incluído, linha a linha, na forma de comentários, intercalados com o texto-objeto gerado pelo compilador.

Uma variável de controle (MIX) pode ser utilizada para controlar a geração desses comentários:

MIX=TRUE: intercala comentários

MIX=FALSE: suprime os comentários

Critérios p/ a geração de código (2)

Para compactar o texto-objeto, a linha de código-objeto é preenchida seqüencialmente pelos códigos da linguagem de saída, mudando-se de linha apenas quando não houver, na linha correntemente em uso, espaço para comportar mais nenhum código.

Uma variável de controle (PACK) pode ser utilizada para controlar esta compactação:

PACK=TRUE: compacta

PACK=FALSE: gera uma instrução por linha

A variável de controle MIX, quando TRUE, faz com que, sempre que um comentário for intercalado, uma nova linha seja utilizada para isto, o que produz uma separação lógica entre os códigos gerados para cada comando do programa-fonte.

Critérios p/ a geração de código (3)

Ao final da geração do código, imediatamente antes de ser gerada a pseudo-instrução END, uma seqüência de pseudo-instruções EXT deverá ser emitida, indicando quais rotinas do ambiente de execução serão necessárias ao funcionamento do programa.

Critérios p/ a geração de código (4)

Uma variável de controle (CODE) pode ser utilizada para controlar a geração de código-objeto:

CODE=TRUE: permite a geração de código

CODE=FALSE: inibe a geração de código

Quando CODE=FALSE, são desativadas as gerações de comentários, e portanto a variável MIX perde sua função. Entretanto, ao ser alterado o valor de CODE, as demais funções são restauradas.

Critérios p/ a geração de código (5)

outra variável de controle (LIST) controla a listagem do programa-fonte:

LIST=TRUE: permite a geração de listagem do texto-fonte

LIST=FALSE: inibe a geração de listagem do texto-fonte

Complementos

Listagem

Na listagem, é impresso o número da linha do texto-fonte, seguido de uma cópia da linha em questão. Eventuais erros de compilação, provenientes da análise daquela linha, são impressos logo a seguir.

A supressão da listagem não deve suprimir as mensagens que reportam os erros detectados pelo compilador.

Em uma implementação desses mecanismos, as variáveis LIST e MIX seriam controladas pela rotina de leitura de programa-fonte: a cada linha lida, consulta-se a variável LIST, imprimindo-se a imagem da linha no dispositivo de saída de listagens caso LIST=TRUE.

Consulta-se a variável MIX, e, caso MIX=TRUE, emite-se uma cópia da linha, na forma de um comentário, no dispositivo de saída do código-objeto.

Controles de Geração de Código

As variáveis PACK e CODE são controladas pelas rotinas de geração de código: a cada vez que uma dessas rotinas é chamada para gerar algum código, consulta-se inicialmente a variável CODE.

Caso CODE=FALSE, nada é gerado.

Caso CODE=TRUE, efetua-se a emissão do código solicitado no dispositivo de saída do código-objeto.

Consulta-se então a variável PACK.

Caso PACK=TRUE, aguarda-se que a linha de código-objeto seja totalmente preenchida antes que seja emitida uma mudança de linha.

Caso PACK=TRUE, gera-se uma nova linha a cada código gerado.

No exemplo a ser detalhado a seguir estas variáveis não serão utilizadas, uma vez que correspondem a um nível de detalhamento do compilador mais refinado que o adotado para o restante do exemplo.

Convém, entretanto, mencionar que os valores dessas quatro variáveis podem ser fornecidos pelo programador ao compilador através da leitura de uma linha especial de controle (por exemplo, a linha que precede a primeira das linhas que compõem o programa a ser compilado).

Analisador Léxico

Projeto

Através da inspeção da gramática BNF da linguagem a ser compilada, pode-se levantar os terminais correspondentes essencialmente às palavras reservadas, sinais de pontuação, letras e dígitos.

Levando-se em conta o grande número de vezes que o analisador léxico é chamado durante a compilação, e a necessidade de utilização de "look-ahead" para a distinção entre identificadores e palavras reservadas.

Ocorre também um grande uso de identificadores e números na gramática, e por isso consideram-se como terminais tanto os identificadores como os números, distinguindo, a posteriori, por consulta a tabela, as palavras reservadas dos identificadores.

Tratamento semelhante pode ser dado aos terminais correspondentes a símbolos compostos.

Gramática Léxica, em Notação de Wirth

(repetida)

```
Tokens não triviais:
identificador, número
```

```
(18) identificador = letra { letra | dígito }.
(19) número = dígito { dígito }.
(20) letra = A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z.
(21) dígito = 0|1|2|3|4|5|6|7|8|9.
```

```
Palavras reservadas:
"END", "LET", "GO", "TO", "OF", "READ", "PRINT", "IF", "THEN", "ELSE"
```

```
Operadores, separadores e outros sinais:
";", ":", "+", "-", "*", "/", "(", ")", ",", ">", "=", "<"
```

```
Símbolos compostos:
":=", "GO" "TO"
```

```
Símbolos filtrados:
Branco, fim de linha, comentário
```

Observações

Brancos, sinais de mudança de linha e comentários inseridos no texto-fonte serão eliminados, e desta maneira o analisador léxico terá o comportamento de um fIltro.

A divisão do texto em seus componentes básicos utilizará o critério do comprimento máximo, ou seja, só será considerado identificado um átomo no instante em que o próximo caractere de entrada, ainda não analisado, não puder ser integrado ao mesmo.

Assim sendo, não há detecção de erros léxicos: todo símbolo que não puder ser acoplado a um átomo será considerado como símbolo inicial do próximo átomo a ser extraído, ou de algum separador.

Para efetuar a análise léxica, será desenvolvido um transdutor finito, cujo alfabeto de entrada é o conjunto de todos os caracteres da linguagem, bem como todos os caracteres que poderão figurar nos comentários, e o seu alfabeto de saída, o conjunto dos átomos relacionados anteriormente.

Representação dos átomos (*tokens*)

Cada átomo será representado por um par de números inteiros, o primeiro correspondente ao tipo de átomo, e o segundo, a uma informação complementar, conforme as convenções mostradas na seguinte tabela:

Terminal	Tipo do Átomo	Informação Complementar
palavra reservada	a própria palavra reservada	irrelevante
identificador	<<identificador>>	índice do identificador na tabela de símbolos
número inteiro	<<número>>	o valor numérico associado ao número
sinal	o próprio sinal	irrelevante
outro símbolo	o próprio símbolo	irrelevante

Observações

Uma tabela de símbolos é utilizada para coletar todos os identificadores declarados no programa, e possui uma correspondência direta com uma tabela de atributos, utilizada pelas ações semânticas.

Pode-se admitir a disponibilidade de rotinas de acesso às tabelas de símbolos e de palavras reservadas.

Uma das rotinas se encarrega de efetuar buscas de identificadores nestas tabelas, retornando a informação do local em que o identificador foi encontrado, ou então a informação de que o identificador não consta na tabela.

Neste caso, outra rotina deve ser utilizada para inserir um identificador na tabela de símbolos.

Para uma primeira implementação, não há necessidade de algoritmos rebuscados, de modo que, para as especificações do projeto em questão, pode-se adotar um método de busca seqüencial, de fácil implementação.

Otimizações do compilador podem alterar, posteriormente, a organização de tais estruturas de dados e os métodos de busca a serem aplicados.

Com as considerações até aqui efetuadas, pode-se conceber o autômato finito que servirá como base para o transdutor que implementa o analisador léxico.

Com as considerações até aqui efetuadas, pode-se conceber o autômato finito que servirá como base para o transdutor que implementa o analisador léxico.

Cada um dos átomos formados por mais de um símbolo pode ser reconhecido através dos seguintes autômatos particulares: identificadores, números e o sinal composto de atribuição.

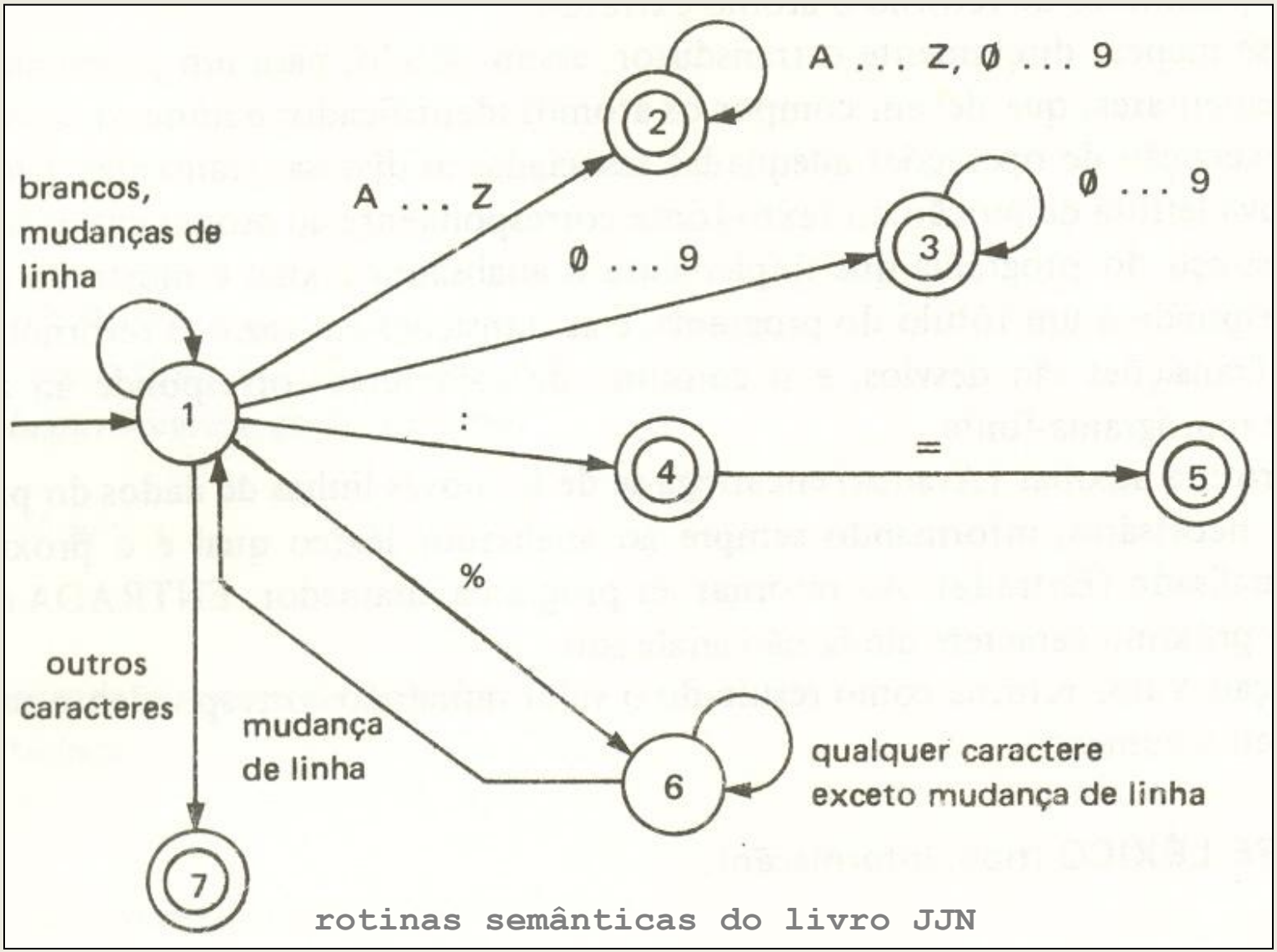
Separadores (comentários, brancos e mudanças de linha) podem ser descartados através de um filtro instalado no estado inicial do autômato que implementará o núcleo do analisador léxico.

Este filtro não deve apresentar estados finais, uma vez que nenhum átomo é reconhecido por ele, ao contrário do que ocorre em outros casos.

Outros símbolos encontrados são tratados isoladamente como átomos.

Compondo-se estes autômatos parciais, obtém-se um reconhecedor-base para o transdutor finito desejado.

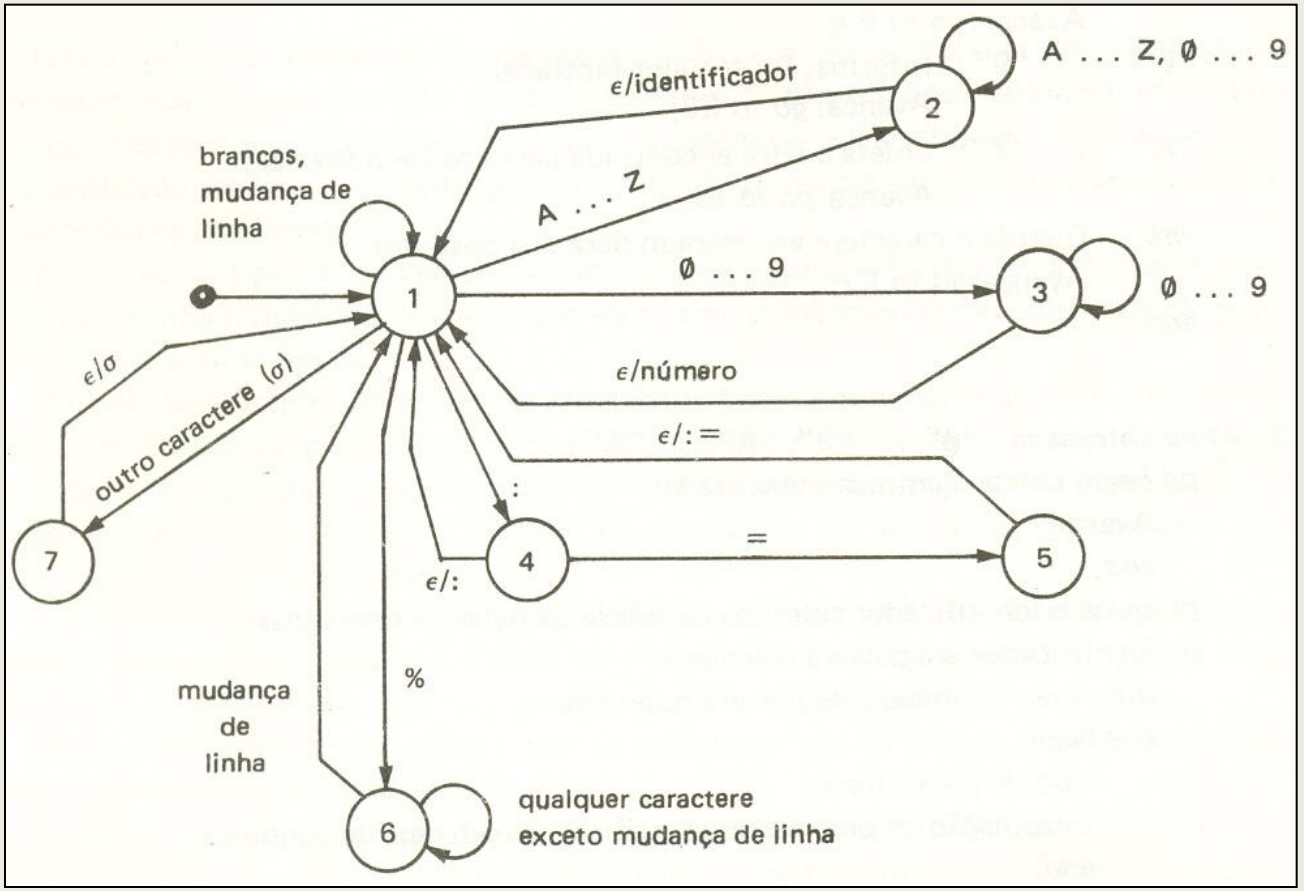
Analisador léxico – diagrama de transições do transdutor finito



rotinas semânticas do livro JJN

Para obter, finalmente, o transdutor, eliminam-se os estados finais, acrescentando-se uma transição em vazio partindo de cada um deles para o estado inicial.

Associa-se a tal transição a emissão do átomo que acaba de ser reconhecido e extraído



Para a operação deste transdutor, convenciona-se que, ao ser executada uma transição em vazio, é efetuado um retorno da subrotina que o implementa para o programa chamador, passando como parâmetro de retorno o átomo extraído.

Pode-se mapear diretamente o transdutor, assim obtido, para a forma de um programa.

As informações complementares, que devem compor os átomos identificador e número, devem ser obtidas através da execução de operações adequadas associadas às diversas transições, para que seja evitada uma nova leitura da porção do texto-fonte correspondente ao átomo extraído.

A seguir, esboça-se a lógica do analisador léxico.

Cada estado corresponde a um rótulo do programa, e cada transição em vazio, a um retorno ao programa chamador.

Transições não-vazias correspondem a desvios condicionais, e o consumo de caracteres corresponde ao avanço de um cursor sobre o programa-fonte, e à extração de um novo caractere.

Uma rotina auxiliar (Avança) encarrega-se de ler novas linhas de dados do programa-fonte sempre que necessário, informando sempre ao analisador léxico qual é o próximo caractere ainda não analisado (Entrada).

Ao retornar ao programa chamador, a variável ENTRADA está sempre se referindo ao próximo caractere ainda não analisado.

A função Valor retorna como resultado o valor numérico correspondente ao dígito fornecido como seu argumento.

O esboço de um programa que implementa o analisador léxico conforme acima convencionado é mostrado a seguir.

Pseudo-código do analisador léxico

```
procedure léxico (tipo, informação);
begin

E1: while entrada = branco or entrada = mudança de linha do
      Avança;
Case entrada of
begin "%" :  Avança; go to E6;
":" :    Avança; go to E4;
"0" : ... "9" : Informação := Valor(entrada); Avança; go to E3;
"A" : ... "Z" : Salva a letra, para uso posterior; Avança; go to E2;
                else: Salva o caractere, para uso posterior;
      Avança; go to E7;
end;

E2: while entrada in { "0", ... , "9", "A", ... , "Z" }
do begin coleta a entrada encontrada; Avança; end;
  pesquisa o identificador coletado na tabela de palavras
      reservadas;
  íf identificador era palavra reservada
     then tipo: = número da palavra reservada
     else begin
 tipo: = identificador
 informação: = posição do identificador na tabela de símbolos
 end;
  return ;
```

```
E3: while entrada in { "0", ... , "9" }
do begin  Informação: = Informação*10 + Valor(entrada);
      Avança; end;
tipo: = número;
return ;

E4: if entrada ≠ "="
then begin tipo := ":"; return; end
else begin Avança; go to E5; end;

E5: tipo := ":="; return ;

E6: while entrada ≠ mudança de linha do Avança; go to E 1 ;

E7: tipo := caractere anteriormente guardado; return;

end léxico;
```

(Esta versão do analisador léxico está projetada para ser usada como subrotina do analisador sintático.)

Analisador Sintático

Pela inspeção da gramática da linguagem, verifica-se que se trata de uma linguagem livre de contexto, mas não regular.

Isto se deve à presença de não-terminais auto-recursivos centrais, tais como <expressão>, <termo>, <fator>, <comando> e <decisão>, o que pode ser facilmente verificado mediante a construção da árvore da gramática ou através de derivações convenientes de cada um dos não-terminais.

Para a construção do reconhecedor sintático desejado, é necessário, antes de mais nada, determinar os não-terminais essenciais, que irão gerar as submáquinas do autômato de pilha em que o reconhecedor se baseia.

Note-se que <expressão>, <termo> e <fator> são mutuamente dependentes, sendo que <termo> só é utilizado para definir <expressão>, e que <fator> só é empregado na definição de <termo>.

Ao contrário, <expressão> é um não-terminal vastamente referenciado por toda a gramática.

Dessa maneira, se <termo> e <fator> puderem ser eliminados por manipulação gramatical, restará apenas o não-terminal <expressão>.

O mesmo raciocínio pode ser feito em relação a <comando> e <decisão>: o conceito de <decisão> só é utilizado para a definição de <comando>, e poderá ser eliminado se houver a possibilidade de aplicação de uma manipulação gramatical apropriada.

Outro não-terminal essencial é, como não poderia ser diferente, a raiz de gramática, a qual deverá dar origem à submáquina principal do autômato de pilha desejado.

Com essas observações em mente, pode-se iniciar o processo de manipulação da gramática, com a finalidade de se obter um reconhecedor sintático que servirá como núcleo do compilador dirigido por sintaxe.

A primeira providência é a de exprimir a gramática através da notação de Wirth, eliminando recursões à direita ou à esquerda, através da conversão de tais recursões em iterações.

O resultado dessa primeira manipulação é o seguinte:

Rotinas semânticas

Gramática do cap.5 do livro Introdução à Compilação

Descrição da Linguagem-exemplo em Notação de Wirth tradicional (Iterativa)

SINTÁTICA

(1) programa = seqüência-de-comandos "END".
(2) seqüência-de-comandos = comando { ";" comando }.
(3) comando = { rótulo ":" } [atribuição | desvio I leitura I impressão I decisão].
(4) atribuição = "LET" identificador ":=" expressão.
(5) expressão = termo { ("+" I "-") termo }.
(6) termo = fator { ("*" I "/") fator }.
(7) fator = identificador I número I "(" expressão ")".
(8) desvio = "GO" "TO" (rótulo I identificador "OF" lista-de-rótulos).
(9) lista-de-rótulos = rótulo { "," rótulo }.
(10) rótulo = identificador.
(11) leitura = "READ" lista-de-Identificadores.
(12) lista-de-Identificadores = [identificador { "," identificador }].
(13) impressão = "PRINT" lista-de-expressões.
(14) lista-de-expressões = [expressão { "," expressão }].
(15) decisão = "IF" comparação "THEN" comando "ELSE" comando.
(16) comparação = expressão operador-de-comparação expressão.
(17) operador-de-comparação = ">" I "=" | "<".

LÉXICA

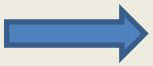
(18) identificador = letra { letra I dígito }.
(19) número = dígito { dígito }.
(20) letra = A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z.
(21) dígito = 0|1|2|3|4|5|6|7|8|9.

rotinas semânticas do livro JJN

programa

programa = comando {";" comando} "END" .

	C	END	;	ε
→ 0	1			
1				2
2		5	3	
3	4			
4				2
5 →				

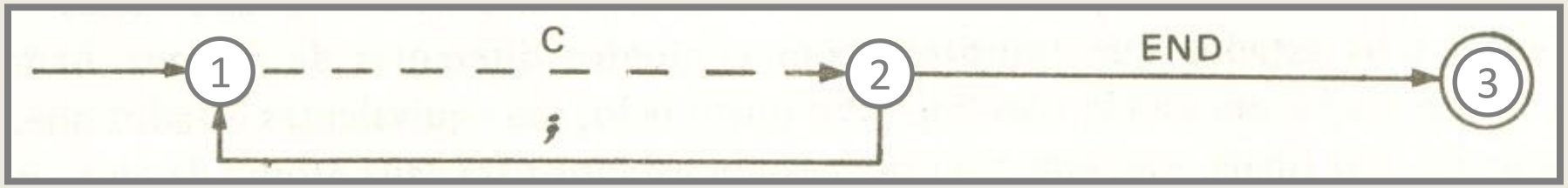


Eliminação de transições em vazio

	C	END	;
→ 0	1		
1		5	3
3	4		
4		5	3
5 →			

Programa - minimizada

	C	END	;
→ 1	2		
2		3	1
3 →			



comando

```
comando =  { identificador ":" }
      [     "LET" identificador ":=" expressão
        I   "GO" "TO" ( identificador [ "OF" identificador { "," identificador} ] )
        I   "READ" [ identificador { "," identificador } ]
        I   "PRINT" [ expressão { "," expressão } ]
        I   "IF" expressão ( "<" I "=" I ">" ) expressão "THEN" comando "ELSE" comando    ].
```

	I	:	LET	:=	E	GOTO	OF	,	READ	PRINT	IF	>	=	<	THEN	C	ELSE
→0→	39		3			7			16	22	28						
3	4																
4				5													
5					6												
6→																	
7	9																
9→							11										
11	12																
12→								14									
14	15																
15→								14									
16→	18																
18→								20									
20	21																
21→								20									
22→																	
24→								26									
26→					27												
27→																	
28					29												
29												33	32	31			
31					34												
32					34												
33					34												
34															35		
35																36	
36																	37
37																38	
38→																	
39		40															
40→	39		3			7			16	22	28						

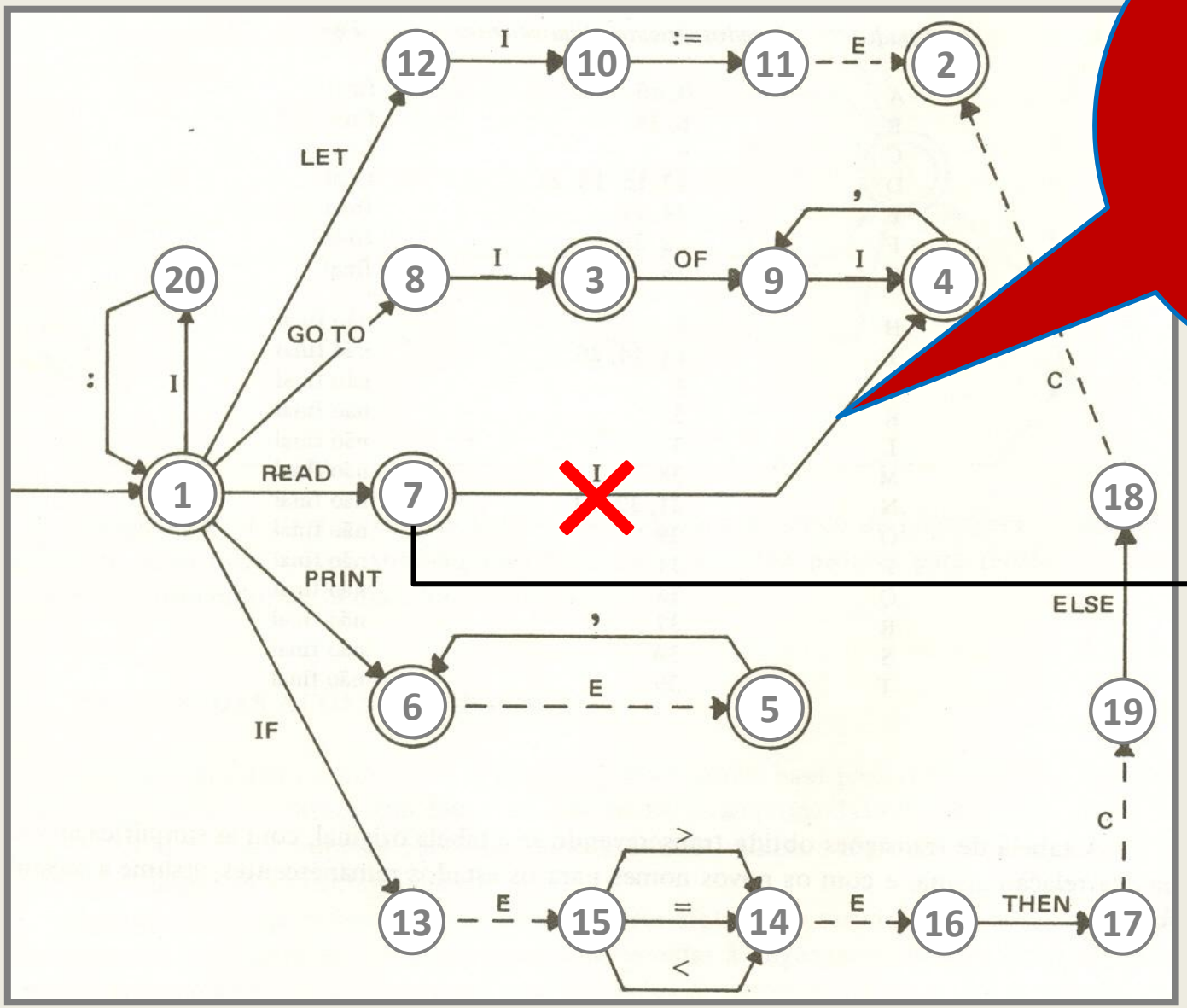
Eliminação de transições em vazio

	I	:	LET	:=	GOTO	OF	,	E	READ	PRINT	IF	>	=	<	THEN	C	ELSE	ε
→0																		1
1	39		3		7				16	22	28							2
2→																		
3	4																	
4				5														
5								6										
6																		2
7	9																	
8																		2
9						11												10
10																		8
11	12																	
12																		13
13							14											10
14	15																	
15																		13
16	18																	17
17																		2
18																		19
19							20											17
20	21																	
21																		19
22								24										23
23																		2
24																		25
25							26											23
26								27										
27																		25
28								29										
29												33	32	31				
30								34										
31																		30
32																		30
33																		30
34															35			
35																36		
36																	37	
37																38		
38																		2
39		40																
40																		1

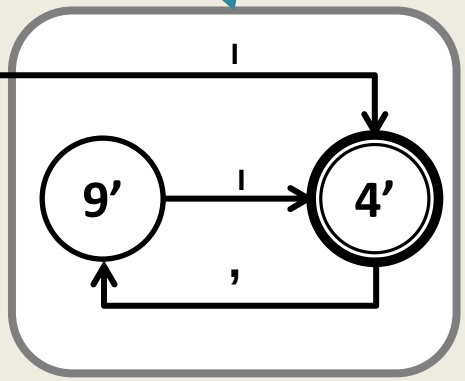
Comando - minimizada

	I	:	LET	:=	E	GOTO	OF	,	READ	PRINT	IF	<	=	>	THEN	C	ELSE
→1→	20		12			8			7	6	13						
2→																	
3→							9										
4→								9									
5→								6									
6→					5												
7→	4																
8	3																
9	4																
10				11													
11					2												
12	10																
13					15												
14					16												
15												14	14	14			
16															17		
17																19	
18																2	
19																	18
20		1															

comando



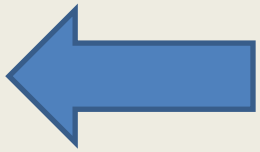
Observação:
apesar de estar sintaticamente correta, esta transição é inconveniente para fins semânticos, e deve ser substituída conforme mostrado no destaque:



expressão

expressão =
(identificador I número I "(" expressão ")")
{ ("*" I "/") (identificador I número I "(" expressão ")") }
{ ("+" | "-") (identificador | número I "(" expressão ")")
{ ("*" I "/") (identificador I número I "(" expressão ")" } } .

	I	N	<	E	>	+	-	*	/
→0→	2	3	4						
2→	20	9	10						
3→	20	9	10						
4				5					
5					6				
6→						19	20	9	10
9	12	13	14						
10	12	13	14						
12→						19	20	9	10
13→	20	9	10						
14				15					
15					16				
16→						19	20	9	10
19	22	23	24						
20	22	23	24						
22→						19	20	29	30
23→						19	20	29	30
24				25					
25					26				
26→						19	20	29	30
29	32	33	34						
30	32	33	34						
32→						19	20	29	30
33→						19	20	29	30
34				35					
35					36				
36→						19	20	29	30



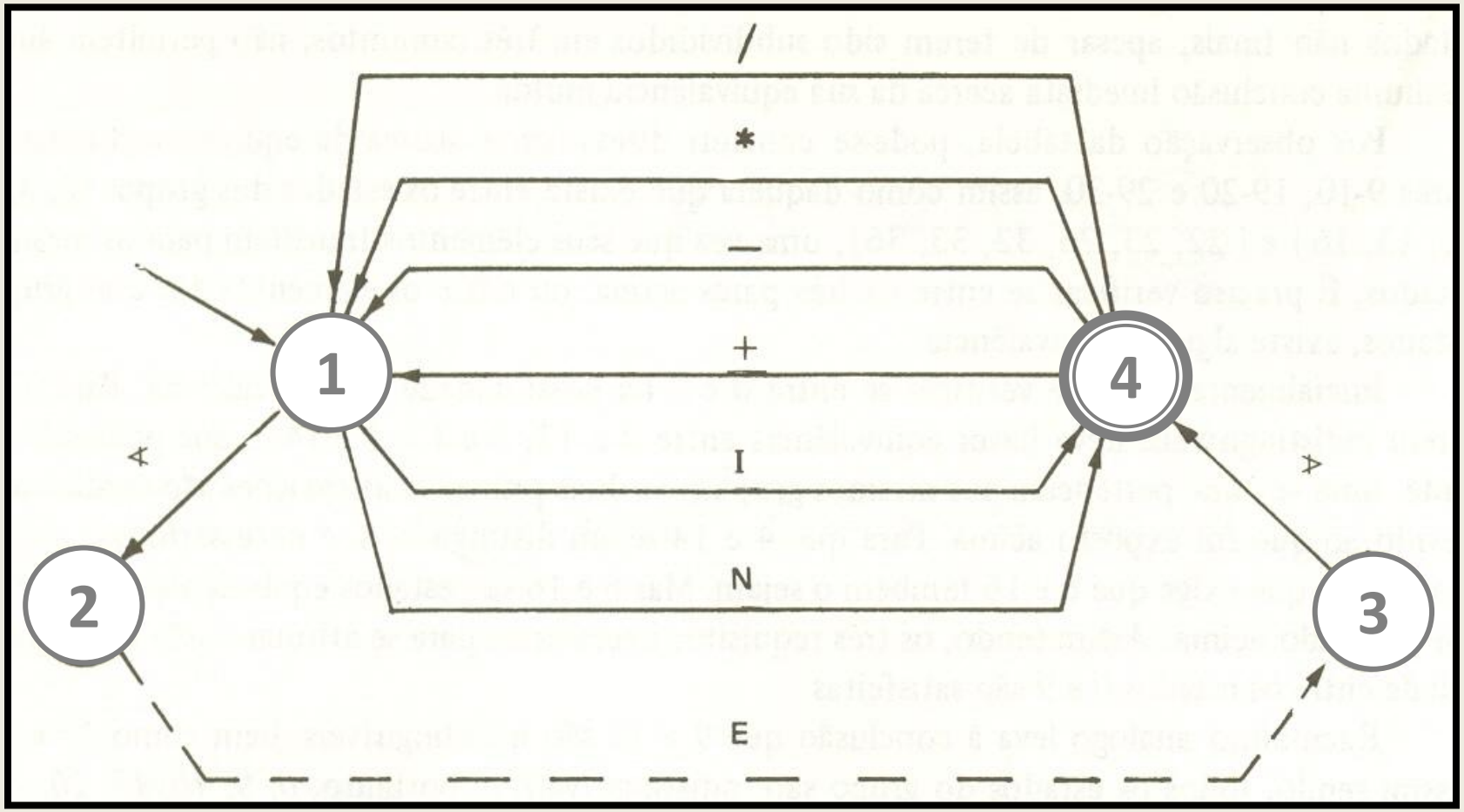
Eliminação de transições em vazio

	I	N	<	E	>	+	-	*	/	ε
→0	2	3	4							
1										7
2										1
3										1
4				5						
5					6					
6										1
7								9	10	17
8	12	13	14							
9										8
10										8
11										7
12										11
13										11
14				15						
15					16					
16										11
17→						19	20			
18	22	23	24							
19										18
20										18
21										27
22						.				21
23										21
24				25						
25					26					
26										21
27								29	30	17
28	32	33	34							
29										28
30										28
31										27
32										31
33										31
34				35						
35					36					
36										31

Expressão - minimizada

	I	N	<	E	>	+	-	*	/
→ 1	4	4	2						
2				3					
3					4				
4 →						1	1	1	1

expressão



Inclusão de rotinas semânticas

- Os autômato agora está pronto para receber as rotinas semânticas, cujas especificações vêm esboçadas a seguir.
- Associe a cada transição do autômato a rotina semântica apropriada, que atenda as especificações indicadas nos slides seguintes.
- Adapte este exemplo ilustrativo, com que estamos trabalhando, ao caso específico da linguagem desenvolvida no projeto.
- Trabalhe de forma incremental, sem pressa, incluindo semântica para uma construção sintática apenas, de cada vez.

Comando vazio

Nada é gerado

Nada é feito

Declarações de rótulos

Identificador$_1$: ... Identificador$_n$: comando

LBL identificador$_1$;
...
LBL identificador$_n$;
Código referente ao* comando*;

Desvio incondicional

GO TO identificador

JP identificador;

Implementação de Rótulos e Comandos de Desvio Incondicional

Usando uma lógica similar à utilizada em montadores de um passo para o tratamento de rótulos, é possível implementar de modo trivial os desvios, gerando instruções de desvio equivalentes para os rótulos correspondentes.

`GO TO X`
`...`
`X:<comando>`
	`JP X`
	`...`
	`X <código para o comando>`

Desvio condicional múltiplo

GO TO identificador$_0$ OF identificador$_1$, ..., identificador$_n$

```
LD    identificador0;
JZ    identificador1;
—     = 1;
...

—     = 1;
JZ    identificadorn;
```

Implementação de Comandos Condicionais

O mais simples deles é o desvio condicional.

Avaliada a condição, ela é testada, desviando para o rótulo apenas no caso de seu valor ser **T**.

GO TO X WHEN A < B	LD A =A - B =A-B JN OK (T porque A<B) LD ZERO (F porque A>=B) JP TEST OK LD UM TEST JZ NEXT (se F, prossegue) JP X (se T, desvia) NEXT <código do próximo comando> ... X <código para o qual o desvio é feito>

If ...then

Outro comando condicional usual é o comando **IF**, que na forma sem a cláusula **ELSE**, avaliada a condição, ela é testada, de forma similar ao desvio condicional, e, caso seja verdadeira, a cláusula **THEN** é executada, caso contrário, desvia-se para o comando seguinte sem nada executar

```
IF A
     = B

         THEN
            READ
                  B
<próximo comando>
```

```
        LD    A      =A
        -     B      =A-B
        JZ    OK     (T porque A=B)
        LD    ZERO   (F porque A#B)
        JP    TEST
OK      LD    UM
TEST    JZ    NEXT   (se F,próx.com.)
        SC    Read   (se T,exec.READ B)
        MM    B
NEXT    <código do próximo comando>
```

If ... then ... else

com a cláusula **ELSE:** avaliada a condição, ela é testada, de forma similar ao desvio condicional, e, caso seja verdadeira, o código correspondente à cláusula **THEN** é executado, seguido de um desvio incondicional para o código referente ao próximo comando, caso contrário, desvia-se para o código correspondente à cláusula **ELSE**, após o qual já se encontra o código do comando seguinte.

```
IF
       A>B


          THEN
             B := A

                ELSE
                     A:=B
<próximo comando>
```

```
        LD    A          =A
        -     B          =A-B
        JN    FALSO      (F porque A<B)
        JZ    FALSO      (F porque A=B)
        LD    UM         (T porque A>B)
        JP    TEST
OK      LD    UM
TEST    JZ    ELSE       (se F, cláusula ELSE)
        LD    A          (se T, executa B:=A)
        MM    B
        JP    NEXT       (ao próximo comando)
ELSE    LD    B          (no ELSE, somente
        MM    A                executa A:=B)
NEXT    <código do próximo comando>
```

Implementação de Comandos Iterativos: While ... do

Há três tipos freqüentes de comandos iterativos:

WHILE – neste comando, o teste é feito ao início da iteração, efetuando-se um desvio para o próximo comando se resultar um valor lógico **F** , e para o início da iteração, na situação inversa.

WHILE A#B	**LOOP**	**LD**	**A**	**=A**
		-	**B**	**=A-B**
		JZ	**TEST**	**(F porque A=B)**
		LD	**UM**	**(T porque A#B)**
	TEST	**JZ**	**NEXT**	**(se F, próximo comando)**
DO READ B		**SC**	**Read**	**(se T, executa READ B)**
		MM	**B**	
		JP	**LOOP**	**(volta a testar se A#B)**
	NEXT	**<código do próximo comando>**		

Do ... until

neste comando, o teste é feito ao final da iteração, efetuando-se um desvio para o início da mesma se desse teste resultar em valor lógico **F** , e desviando para o comando seguinte, na situação inversa.

DO READ B	LOOP	SC	Read	(executa READ B)
		MM	B	
UNTIL		LD	A	=A
		-	B	=A-B
		JZ	TEST	(F porque A=B)
A#B		LD	UM	(T porque A#B)
	TEST	JZ	LOOP	(se F, próximo comando)
		<código do próximo comando>		

For ... Step ... Until ... Do

Neste tipo de comando, uma variável aritmética é utilizada para controlar a evolução da iteração. O comando especifica um valor inicial para essa variável, e todas as vezes que o corpo da iteração reinicia sua execução, o incremento especificado é adicionado à variável de controle, a qual é testada em seguida contra o valor final também especificado no comando. Note-se que o valor inicial, do incremento e final da variável de controle pode ser, cada um deles, especificado na forma de expressão aritmética, portanto tais expressões devem ser reavaliadas a cada iteração realizada.

```
FOR I:=1                    LD    UM
                            MM    I
                            JP    DO
    STEP 2                  LD    I
                            +     DOIS
                            MM    I
        UNTIL 50            -     50
                            JN    DO
                            JP    NEXT
           DO               LD    I
               PRINT I      SC    Print
                            JP    LOOP
                            NEXT  <código do próximo comando>
```

Atribuição

LET identificador := expressão

...
***código-objeto referente à* expressão**
(retorna o resultado no acumulador) ;

MM identificador;

...

Atribuição múltipla

LET identificador$_1$:= ... := identificador$_n$

:= expressão

***código-objeto referente à* expressão**
(retorna o resultado no acumulador) ;

MM identificador$_1$;

...

MM identificador$_n$;

Implementação dos Comandos de Atribuição de Valor

- Para implementar qualquer comando de atribuição de valor, gera-se primeiramente o código da expressão cujo valor se deseja atribuir.
- Esse código calcula no acumulador o valor da expressão. Segue-se a atribuição, depositando o conteúdo do acumulador no destino indicado à esquerda do sinal **:=**, no comando de atribuição.
- No exemplo a seguir, observe-se que as instruções posteriores à chamada **SC FF** referem-se à expressão aritmética e não à chamada de **FF**, e que a instrução **MM B** efetua a atribuição propriamente dita.

```
B:=A                  LD   A
    -FF               SC   Push    pilha=A.
       ( X,           LD   X
                      SC   Push    pilha=X;A.
          Y           LD   Y
           +Z         +    Z       =Y+Z
                      SC   Push    pilha=Y+Z;X;A.
              )       SC   FF      pilha=A.
                      MM   TEMP1   =FF(X,Y+Z)
                      SC   Pop     =A
                      -    TEMP1   =A-FF(X,Y+Z)
                      MM   B       B recebe A-FF(X,Y+Z)
```

Implementação das Expressões Lógicas (Booleanas)

Convencionando-se que o valor **0** (zero) represente o valor lógico **FALSE**, e que o valor **1** (um) represente o valor lógico **TRUE**, basta deixar no acumulador um desses dois valores ao final da avaliação de um valor lógico para que o cálculo de expressões lógicas se torne análogo ao das expressões aritméticas (alguns exemplos do código gerado para expressões lógicas simples podem ser vistos nos comandos condicionais, apresentados adiante). Operadores lógicos poderão ser implementados com o uso das operações aritméticas associadas: **AND** através de produto (*) e **OR** através de soma (+). A complementação **NOT** não é imediata, devendo ser sintetizada, por exemplo, como ilustrado acima. As posições **UM** e **M1** devem conter os valores **1** e **–1**, respectivamente, e devem ter sido definidas no ambiente de execução.

```
NOT(A AND
          (B
             OR C))
```

```
LD    A
SC    Push     pilha=A.
LD    B
+     C        =B+C
JZ    *+2
LD    UM
MM    TEMP1    =(B OR C)
SC    Pop      =A
*     TEMP1    =A*(B OR C)
JZ    *+2
LD    UM       =A AND(B OR C)
JZ    *+2
LD    M1
+     UM       =NOT(A AND(B OR C))
```

Leitura sem parâmetros

READ

SC #READ;

Leitura de lista de variáveis

READ identificador$_1$, ... , identificador$_n$

```
SC      #READ;
MM      identificador1;
SC      #READ;
MM      identificador2;
...

SC      #READ;
MM      identificadorn;
```

Comandos de Entrada de Dados (Aritméticos/Lógicos)

complementa as atividades de entrada/saída aritmética a implementação dos comandos de entrada de dados aritméticos. Os dados lidos para o acumulador por uma rotina de leitura (**Read**) do ambiente de execução podem ser depositados em variáveis do programa (**X**). Procede-se analogamente para a saída de dados lógicos.

READ X	**SC Read (chama a função)** **MM X (guarda result. na var.X)**

Impressão sem parâmetros

PRINT

SC #NEWLINE;

Impressão de lista de valores

PRINT expressão$_1$, ... , expressão$_n$

código relativo à* expressão$_1$*;
SC #CONVERT;
SC #NEWLINE;
...

código relativo à* expressão$_n$*;
SC #CONVERT;
SC #NEWLINE;

Comandos de Saída de Dados Aritméticos

Disponibilizada a geração do código para expressões aritméticas, e havendo no ambiente de execução uma rotina (**Print**) que imprima um argumento presente no topo da pilha, pouco deve ser adicionado para a implementação de comandos de saída de dados aritméticos, o qual, por isso, é aqui sugerido como passo seguinte desta implementação.

Comandos de Saída de Dados Aritméticos

Disponibilizada a geração do código para expressões aritméticas, e havendo no ambiente de execução uma rotina (**Print**) que imprima um argumento presente no topo da pilha, pouco deve ser adicionado para a implementação de comandos de saída de dados aritméticos, o qual, por isso, é aqui sugerido como passo seguinte desta implementação.

PRINT (G - H)	**LD G (=G) - H (=G-H) SC Push (argumento => pilha) SC Print (chama a função)**

Comandos de Saída de Dados Lógicos

Estando no acumulador um valor lógico, representado pelos valores aritméticos **1** ou **0**, o comando de saída deve converter respectivamente esses valores para os caracteres **T** e **F**, antes de efetuar a saída do dado. As posições **T** e **F**, declaradas no ambiente de execução, devem conter os padrões ASCII referentes às letras **T** e **F**, respectivamente.

PRINT X	**LD X** **JZ LDF** **LD T** **JP SCP** **LDF LD F** **SCP SC Push** **SC Print**

Decisão (if-then-else)
1 – comparação de expressões

IF expressão$_1$ comparação expressão$_2$

THEN comando$_1$ ELSE comando$_2$

código referente à* expressão$_1$*;

MM #TEMP;

código referente à* expressão$_2$*;

— #TEMP;

(Continua...)

Decisão (if-then-else)
2 - Operações de comparação

Comparação > :

```
JZ    #ELSE_k;
JN    #ELSE_k;
JP    #THEN_k;
                (Continua...)
```

Comparação =:

```
JZ    #THEN_k;
JP    #ELSE_k;
                (Continua...)
```

Comparação <:

```
JN    #THEN_k;
JP    #ELSE_k;
                (Continua...)
```

Decisão (if-then-else)
3 - comandos e desvios associados

LBL #THEN$_k$;

***código referente ao* comando$_1$;**

JP #FIM$_k$;

LBL #ELSE$_k$;

***código referente ao* comando$_2$;**

LBL #FIM$_k$;

Decisão Múltipla: Case ... of

Comandos de decisão múltipla podem ser compilados como se representassem uma seqüência de comandos de decisão simples `IF ... THEN`, cada qual seguido por um desvio incondicional para o comando seguinte ao de decisão múltipla.

A cada teste especificado, o código associado avalia se seu valor lógico é `F`, desviando para o teste seguinte se isso acontecer, e permitindo a execução do comando correspondente se o teste resultar em um valor lógico `T`.

Após o código do comando associado a cada um desses testes, um desvio incondicional (`JP NEXT`) força a execução do comando que segue o comando de decisão múltipla.

A cláusula `ELSE` especifica o comando a ser executado na eventualidade de todos os testes falharem, e o código a ela associado é similar à das situações anteriores, exceto pela ausência do desvio forçado `JP NEXT`, que aqui é desnecessário.

```
CASE N  OF {
 1:                                 LD    N         (teste do caso 1)
                                    -     K1
                                    JZ    CASE1     (cláusula 1 porque N=1)
                                    JP    SEG1      (caso seguinte porque N#1)
      I:=X;                 CASE1   LD    X         (executa cláusula 1)
                                    MM    I
                                    JP    NEXT      (desvia ao próximo comando)
 3:                         SEG1    LD    N         (teste do caso 3)
                                    -     K3
                                    JZ    CASE3     (cláusula 3 porque N=3)
                                    JP    SEG3      (caso seguinte porque N#3)
      READ X;               CASE3   SC    Read      (executa cláusula 3)
                                    MM    X
                                    JP    NEXT      (desvia ao próximo comando)
 25:                        SEG3    LD    N         (teste do caso 25)
                                    -     K25
                                    JZ    CASE25    (cláusula 25 porque N=25)
                                    JP    SEG25     (caso seguinte porque N#25)
      PRINT X;              CASE25  LD    X         (executa cláusula 25)
                                    SC    Print
                                    JP    NEXT      (desvia ao próximo comando)
 ELSE                       SEG25   SC    Read      (não há mais cláusulas =>
      READ B   }                    MM    B                executa cláusula ELSE)
<próximo comando >          NEXT    <código do próximo comando>
```

INSERÇÃO DAS ROTINAS SEMÂNTICAS NOS AUTÔMATOS

Observações

- Nos slides seguintes, as rotinas semânticas referentes a cada submáquina serão indicadas por meio de um nome numérico (p/ex. 1, 2, 3, etc.), e sua associação às transições do autômato serão denotadas na tabela de transições incluindo-se o número da rotina semântica à direita do estado-destino nas células da tabela de transições, separados por barra.
- Embora constituam esboços bastante completos, as rotinas semânticas aqui apresentadas em pseudo-código obviamente não estão prontas para serem utilizadas, ficando como exercício sua implementação definitiva na linguagem de programação escolhida para a codificação do projeto.

Expressões
1 – operações aritméticas

Topo da Pilha de operadores	Topo da Pilha de operandos	2ª posição da pilha de operandos	código-objeto a ser gerado
+	A	B	LD B; + A; MM #TEMP$_i$;
-	A	B	LD B; - A; MM #TEMP$_i$;
*	A	B	LD B; * A; MM #TEMP$_i$;
/	A	B	LD B; / A; MM #TEMP$_i$;

Expressões
2 – tabela de transições (ilustração)

A título de exemplo, mostra-se abaixo a tabela de transições da sub-máquina de Expressões. Nos slides seguintes, detalham-se os pseudo-códigos das rotinas semânticas a ...m, as quais, naturalmente, estão ainda por codificar.
A seta à esquerda de 1 indica que1 é o estado inicial da submáquina.
A seta à direita de 4 indica que 4 é um estado de aceitação da submáquina.
As ações de saída são executadas no caso de ser atingida uma célula vazia.
A ação j não fornece mensagem de erro, pois 4 é um estado de aceitação.

	l	N	<	E	>	+	-	*	/	Ação de Saída
→ 1	4/a	4/b	2/c							k
2				3/d						l
3					4/e					m
4 →						1/f	1/g	1/h	1/i	j

Expressões
3 – Rotinas semânticas (1ªparte)

a	empilha (pilha de operandos, identificador encontrado)
b	empilha (pilha de operandos, número encontrado)
c	empilha (pilha de operadores, "(")
d	nada executa
e	X5: consulta (pilha de operadores, Y);
	Se Y ≠ "(" : executa GERACÓDIGO, detalhada adiante; GO TO X5;
	Se Y = "(" : desempilha (pilha de operadores, Y);
f	X6: consulta (pilha de operadores, Y);
	Se Y for "+", "-", "*" ou "/":
	executa GERACÓDIGO, detalhada adiante; GO TO X6;
	Caso contrário: empilha (pilha de operadores, "+");
g	X7: consulta (pilha de operadores, Y);
	Se Y for "+", "-", "*", ou "/":
	executa GERACÓDIGO, detalhada adiante; GO TO X7;
	Caso contrário: empilha (pilha de operadores, "-");

Expressões
4 – rotinas semânticas (2ªparte)

h	X8: consulta (pilha de operadores, Y);
	Se Y for "*" ou "/": executa GERACÓDIGO, detalhada adiante; GO TO X8;
	Caso contrário: empilha (pilha de operadores, "*");
i	X9: consulta (pilha de operadores, Y);
	Se Y for "*" ou "/": executa GERACÓDIGO, detalhada adiante; GO TO X9;
	Caso contrário: empilha (pilha de operadores, "/");
j	(* Esta rotina é associada ao final do reconhecimento da expressão *)
	X10: consulta (pilha de operadores, Y);
	Se Y não for "1": executa GERACÓDIGO, detalhada adiante, GO TO X10;
k	ERRO ("esperava-se identificador, número ou '(' neste ponto").
l	ERRO ("esperava-se uma expressão correta neste ponto").
m	ERRO ("esperava-se ')' neste ponto").

Expressões
5 – rotinas auxiliares (1ªparte)

GERACÓDIGO:

n	Desempilha (pilha de operadores, Y);
o	Desempilha (pilha de operandos, B);
p	Desempilha (pilha de operandos, A);
q	Gera ("LD", A);
r	Se Y = "+": gera (“+", B);
s	Se Y = "-": gera (“-", B);
t	Se Y = "*": gera (“*", B);
u	Se Y = "/": gera (“/", B);
v	Incrementacontador (CONTATEMP);
w	Gera (“MM", "#TEMP", CONTATEMP);
x	empilha (pilha de operandos, "#TEMP", CONTATEMP);

Implementação de expressões aritméticas

```
A
 +(
   B
    -(
      C
       -D
         )
          *E
            )
             +
                G
                 *(
                    H
                     -I
                        )
```

```
LD   A
SC   Push    pilha=A.
LD   B
SC   Push    pilha=B;A.
LD   C
-    D
MM   TEMP1   =(C-D)   (não será usado)
*    E
MM   TEMP2   =(C-D)*E
SC   Pop     =B
-    TEMP2   =B-(C-D)*E
MM   TEMP3   =B-(C-D)*E
SC   Pop     =A
+    TEMP3   =A+(B-(C-D)*E)
SC   Push    pilha=A+(B-(C-D)*E).
LD   F
SC   Push    pilha=G;A+(B-(C-D)*E).
LD   H
-    I
MM   TEMP4   =(H-I)
SC   Pop     =G
*    TEMP4   =G*(H-I)
MM   TEMP5   =G*(H-I)
SC   Pop     =A+(B-(C-D)*E)
+    TEMP5   =A+(B-(C-D)*E)+G*(H-I)
```

Usar como base um reconhecedor de expressões aritméticas como o estudado em aula. Lembrar da conveniência da conversão intermediária para notação polonesa, para facilitar a geração de código, e da utilidade de uma pilha para guardar temporariamente o código incompleto, que estiver sendo gerado ao ser iniciada cada nova expressão aninhada. Assim, uma expressão como **a+(b-(c-d)*e)+f*(g-h)** pode ser compilada diretamente, usando uma pilha auxiliar como apoio, ou então convertida antes para notação polonesa, a partir da qual o código final pode ser facilmente gerado. De qualquer maneira, o resultado do cálculo deve ser deixado no acumulador. Existindo rotinas de simulação de pilha (**Push** e **Pop**) no ambiente de execução, o código gerado deverá resultar semelhante ao acima apresentado.

No entanto, pode ser mais conveniente gerar código para uso direto do hardware da MVN. Nesse caso, é preciso adaptar esse procedimento, conforme visto em aula, de forma que o código seja gerado adequadamente. Ficam a cargo do aluno essas adaptações, que não apresentam nenhuma dificuldade, mas exigem uma atenção maior ao gerenciamento das variáveis temporárias necessárias para o armazenamento de valores intermediários calculados durante esse processo.

Implementação de Chamadas de Função Aritmética

Gera-se código para as funções aritméticas paramétricas construindo um código que avalie os argumentos e os empilhe um a um, seguindo-se uma instrução de chamada da função desejada.

O resultado deve ser depositado no acumulador.

FF **(X,** **Y** **+Z** **)**	**LD X** **SC Push pilha=X.** **LD Y** **+ Z =Y+Z** **SC Push pilha=Y+Z;X.** **SC FF pilha=.**

Implementação de Chamadas de Função Lógica

gera-se código para funções lógicas paramétricas de forma idêntica ao que foi feito para funções aritméticas, construindo-se inicialmente um código que avalie cada argumento empilhando-os um a um, e finalizando com uma instrução de chamada da função lógica desejada. O resultado do cálculo da função deve restar no acumulador.

```
F
  (X,
      Y
       +Z
         )
```

```
LD    X
SC    Push     pilha=X.
LD    Y
+     Z        =Y+Z
SC    Push     pilha=Y+Z;X.
SC    F        pilha=.
```

Implementação de Comandos de Chamada de Subrotina

Muito similar à das funções aritmética e lógica, mas com a diferença de que subrotinas não calculam resultados nem deixam valores no acumulador na ocasião do término da sua execução.

Dessa forma, chamadas de subrotinas não podem ser usadas em expressões, devendo ser consideradas estritamente como comandos imperativos usuais.

```
CALL SUB
  (X,
     Y
      +Z
        )
```

```
LD   X
SC   Push    pilha=X.
LD   Y
+    Z       =Y+Z
SC   Push    pilha=Y+Z;X.
SC   SUB     pilha=.
```

Geração de Código para Declarações de Subrotinas

Muito similar à dos programas, exceto quanto à presença da seção de parâmetros e do retorno ao programa chamador

```
SUBROUTINE  SUB
 (A,                  SC    Pop     Move primeiro argumento
                      MM    A       para o parâmetro formal A
    B                 SC    Pop     Move segundo argumento
     )                MM    B       para o parâmetro formal B
    . . .             . . .
       RETURN         SR            Retorna ao programa chamador
```

Geração de Código para Declarações de Funções

O código para funções é igual ao das subrotinas, mas deve deixar no acumulador o valor de retorno calculado, para uso na expressão na qual a função foi chamada.

Uma verificação deve ser feita quanto à presença ou não, no corpo de uma função, de comandos que efetivamente calculem e retornem ao programa chamador o valor calculado.

É preciso ainda assegurar que de fato tenha sido informado qual resultado deve ser retornado pela função ao programa chamador.

```
FUNCTION F
 (A,

    B
     )
<Calcula valor>
      F:=valor
RETURN
```

```
F DS   0       para endereço de retorno
  SC   Pop     Move primeiro argumento
  MM   A       para o parâmetro formal A
  SC   Pop     Move segundo argumento
  MM   B       para o parâmetro formal B
  . . .
  LD   valor   resultado no acumulador
  SR   F       Retorna ao prog. chamador
```

Programa
1 – código principal

comando$_1$; ... comando$_n$ END

código referente ao* comando$_1$*;
...
código referente ao* comando$_n$*;
SC #STOP;

Programa
2 – área das variáveis declaradas

```
LBL identificador1;
DS      0;
LBL identificador2;
DS      0;
...
LBL identificadorn;
DS      0;
```

Programa
3 – área dos temporários utilizados

```
LBL     #TEMP;
DS      0;
LBL     #TEMP1;
DS      0;
LBL     #TEMP2;
DS      0;
...

LBL     #TEMPn;
DS      0;
```

Programa
4 – rotinas externas e fim do programa

```
EXT     #NEWLINE;
EXT     #CONVERT;
EXT     #READ;
EXT     #STOP;
END;
```

Construção das Rotinas do Ambiente de Execução

Um simples levantamento das variáveis, constantes e rotinas de biblioteca do ambiente de execução chamadas nesses exemplos dá uma boa idéia do conteúdo do ambiente de execução necessário para esse compilador.

ASPECTOS DE IMPLEMENTAÇÃO E INTEGRAÇÃO

O programa principal se limita a executar três procedimentos, em seqüência:

```
Programa principal: iniciação;
                    análise sintática (PROGRAMA);
                    finalização;
```

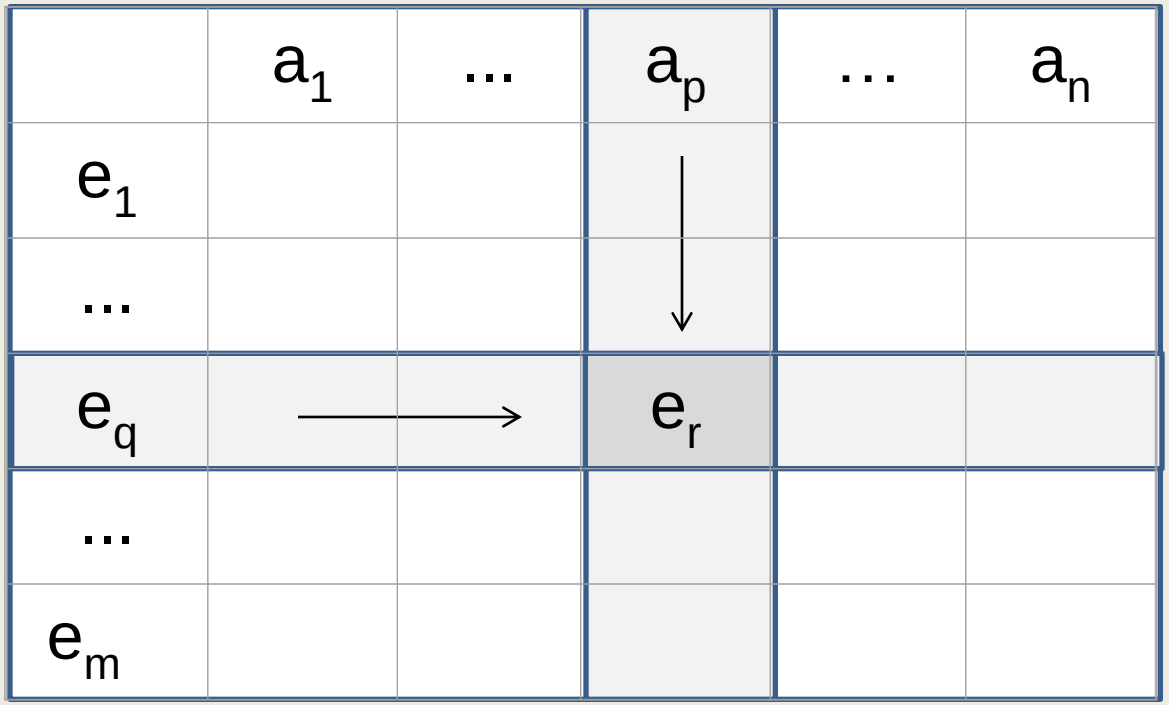
Bases de dados utilizadas

SUBMÁQUINA	**indica a submáquina correntemente em uso.**
ESTADO	**indica o estado corrente da submáquina em uso**
PILHA SINTÁTICA	**estrutura organizada em pilha, cujos elementos são pares (submáquina, estado) e registram os estados de retorno durante o reconhecimento sintático.**
TOPO SINTÁTICA	**aponta para o elemento de PILHA SINTÁTICA mais recentemente colocado nesta estrutura (topo da pilha)**
TRANSIÇÕES PROGRAMA	**tabela de transições da submáquina correspondente ao não-terminal programa.**
TRANSIÇÕES EXPRESSÃO	**idem, expressão**
TRANSIÇÕES COMANDO	**idem, comando**
AÇÕES PROGRAMA	**tabela de ações semânticas correspondentes a TRANSIÇÕES PROGRAMA**
AÇÕES EXPRESSÃO	**idem, TRANSIÇÕES EXPRESSÃO**
AÇÕES COMANDO	**idem, TRANSIÇÕES COMANDO**
ÁTOMOS PROGRAMA	**vetor de átomos e submáquinas com que transita TRANSIÇÕES PROGRAMA**
ÁTOMOS EXPRESSÃO	**idem, TRANSIÇÕES EXPRESSÃO**
ÁTOMOS COMANDO	**idem, TRANSIÇÕES COMANDO**

Códigos associados aos não-terminais

código	submáquina
1	PROGRAMA
2	EXPRESSÃO
3	COMANDO

Tabelas de transições



Codificação das células

ZERO	**indica que não há transição prevista nesta célula. Se o estado correspondente for estado final, indica que deve ser efetuado um retorno para a submáquina chamadora, ou então que terminou o processamento. Para estados não finais, isto corresponde a um erro de sintaxe.**
POSITIVO	**o número contido na célula é interpretado como o número do estado para onde a submáquina deve evoluir nesta transição.**

Pseudo-código do Reconhecedor Sintático

```
Reconhecimento Sintático (S):
        Salvar inicialmente ESTADO e SUBMÁQUINA na PILHA SINTÁTICA:
        Fazer ESTADO: =1; SUBMÁQUINA: = S;
LOOP:   Chamar o Analisador LÉXICO (TIPO, INFORMAÇÃO);
        Buscar TIPO entre os átomos desta submáquina, associando-a assim à coluna da tab.de transições;
        (* se COLUNA = 0, isso indica que não encontrou o átomo *)
        Obter CÉLULA: = tabela de transições corrente [ESTADO, COLUNA]
        Se COLUNA ≠ 0
            então Se CÉLULA ≠ 0
                        então executar rotina indicada em tabela de ações semânticas [ESTADO, COLUNA];
                              fazer ESTADO: = CÉLULA;
                              executar a ação sintática associada à transição, se existir
                        caso contrário,  Se tabela de ações semânticas [ESTADO, 0] indicar um estado final,
                                              então retornar (* à submáquina chamadora *)
                                              se não, emitir mensagem de erro e terminar o processamento.
            se não (*COLUNA = 0),
                        não consumir o átomo;
                        se o estado for final, então retornar à submáquina chamadora.
                                               se não, emitir mensagem de erro e terminar o processamento.
```

FIM

Roteiro de Implementação

A ser preenchido, para cada implementação a ser desenvolvida, de acordo com as particulares necessidades de cada projeto.

Ferramentas de apoio
1. Para autômatos finitos

- Simulador geral de autômatos finitos
- Extrator de gramática léxica a partir da gramática da linguagem de programação
- Gerador de autômatos finitos a partir de gramáticas léxicas
- Biblioteca de ações semânticas para análise léxica
- Ferramenta para inserção de ações semânticas
- Linguagem para definir ações semânticas e associar ações de biblioteca às transições do autômato finito

Ferramentas de apoio
2. Para autômatos de pilha estruturados

- Gerador de autômatos de pilha estruturados a partir da gramática da linguagem de programação
- Biblioteca de ações semânticas para análise sintática
- Biblioteca de ações semânticas para geração de código
- Ferramenta para inserção de ações semânticas
- Linguagem para definir ações semânticas de geração de código e associar ações de biblioteca às transições do autômato de pilha estruturado
- Simulador geral de autômatos de pilha estruturados

Ferramentas de apoio
3. Para a execução

- Biblioteca do ambiente de execução