UNIP
UNIVERSIDADE PAULISTA
Interativa

Compiladores e Computabilidade

Sumário

Compiladores e Computabilidade

APRESENTAÇÃO

Mesmo compreendendo os vários aspectos funcionais que envolvem os sistemas computacionais, a ideia de criar um programa que seja capaz de transformar as instruções de um código-fonte em um programa executável nos parece uma tarefa desafiadora. Esta disciplina demonstrará que, reunindo os conhecimentos de diversas subáreas da computação e adotando uma metodologia apropriada, essa tarefa pode ser realizada e com sucesso!

Embora esta disciplina possa ser conduzida de modo prático, o projeto e a construção de um compilador propriamente dito são mais adequados quando se busca uma formação mais detalhada e aprofundada, típica em um curso de pós-graduação. Aqui, a intenção é de que o aluno compreenda: quais são as tarefas envolvidas no processo de transformação dos algoritmos codificados em programas, a maneira como as informações relevantes são extraídas e como podem ser compreendidas de maneira adequada e, por fim, como criar uma sequência funcional análoga utilizando o conjunto de instruções de máquina e chamadas de sistema, considerando os aspectos da arquitetura escolhidas.

Ao término do curso, o aluno compreenderá melhor diversos aspectos fundamentais em uma linguagem de programação, relacionará estruturas e conjuntos de comandos com seus correlatos em termos de *hardware*, além de ter visto um exemplo consistente de um projeto modular e integrado.

Objetivos gerais

A construção de compiladores é um ramo da ciência da computação cujo estudo agrega conhecimentos e habilidades importantes aos profissionais dessa área, tais como: um exemplo de estruturação apropriada do problema, abordagem metodológica consistente e necessária ao desenvolvimento de projetos desse porte, além de experiências em sistemas que congregam diferentes conhecimentos específicos de computação.

Dadas as características de interpretação e tradução de informações, o estudo dos conceitos fundamentais envolvidos no projeto e na construção de compiladores permite uma aplicação mais ampla das técnicas envolvidas, uma vez que torna o aluno familiarizado com os elementos necessários à escrita de interpretadores de comandos e programas de interface, bem como para o processamento de dados estruturados e extensíveis.

Objetivos específicos

Familiarizar os alunos com as principais técnicas e conceitos envolvidos na compilação de programas de computador. Aprimorar seus conhecimentos sobre programação por meio do estudo da estrutura e das características de uma linguagem de programação, bem como das tarefas realizadas por um compilador para transformar os programas em seus equivalentes em linguagem de máquina.

Conhecer a organização e as operações básicas de um compilador. Compreender o impacto das características e dos paradigmas das diferentes linguagens de programação. Relacionar os fundamentos de programação com as tarefas desempenhadas pelo *hardware*, sendo capaz de compreender melhor

a evolução das novas arquiteturas de computador. Capacitá-lo no desenvolvimento de ferramentas de geração de programas, para avaliação e apoio aos processos de engenharia de *software*, interpretadores e processadores de dados estruturados, bem como no desenvolvimento de novas linguagens e paradigmas de programação.

Unidade I

1 INTRODUÇÃO

Para estudarmos os conceitos e as técnicas que envolvem a construção de compiladores, convém tentar responder algumas perguntas importantes logo de início:

O que ocorre desde a codificação de um algoritmo por um programador até a sua execução em um computador?

Quais são os elementos envolvidos? O que compete a cada um deles nesse processo de transformação?

1.1 Da codificação à execução de um programa

Podemos descrever o processo que transforma um programa fonte em um programa executável por meio de uma sequência encadeada de atividades, cada qual realizada por um elemento específico. Esse processo pode ser ilustrado (figura 1) da seguinte maneira:

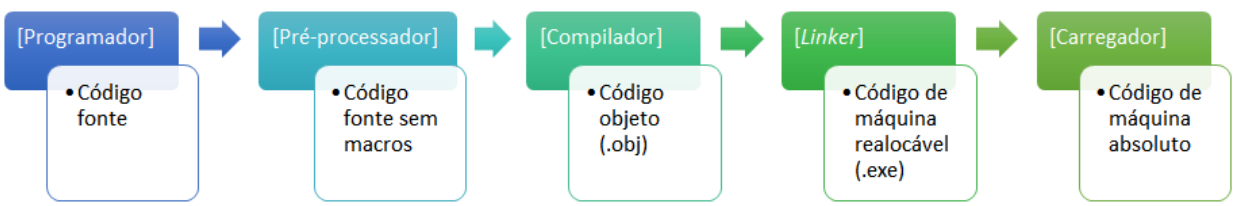

Figura 1 - Diferentes etapas e produtos que envolvem o processo de compilação e execução, desde a sua codificação até a obtenção de um programa possível de ser executado pela máquina

No início do processo, o programador descreve o algoritmo utilizando uma linguagem específica de programação, salvando os comandos e as instruções em um arquivo no formato texto. A linguagem utilizada pelo programador é chamada de linguagem fonte e o conteúdo do arquivo é conhecido como código-fonte.

Em muitos casos, antes de ser traduzido pelo compilador, o código é submetido a uma etapa de pré-processamento em que são realizadas tarefas como a substituição de macros e remoção de diretivas que funcionam como meta-instruções, isto é, informações que orientam o processo de tradução, mas que não fazem parte efetivamente do algoritmo.

Ainda bastante parecido com o código escrito pelo programador, mas sem macros e metainstruções, o código-fonte pode então ser passado ao compilador para ser transformado em código de máquina (código objeto), de acordo com a plataforma alvo.

Nesse novo arquivo criado pelo compilador, habitualmente nomeado com extensão *obj*, temos apenas o código referente às instruções e às funções definidas por esse programador, faltando ainda juntá-lo com o código de máquina relativo às funções das bibliotecas. Para que isso seja feito, um programa realiza a ligação entre os diversos arquivos objetos e acrescenta os cabeçalhos necessários ao sistema operacional para identificá-lo como um arquivo executável (código de máquina realocável).

Quando desejamos executar o programa, o carregador ou *loader* que é considerado parte do sistema operacional aloca o código em alguma porção de memória que esteja disponível. Nesse momento, os endereços das variáveis estáticas e de entrada de função que até o momento estavam sob a forma de deslocamentos são calculados e resolvidos. É por esse motivo que apenas depois de ser colocado em memória é que o programa é chamado de código de máquina absoluto.

1.2 Caracterização das linguagens quanto ao nível de abstração

Existem inúmeras linguagens de programação e, certamente, surgirão outras novas daqui para frente. É certo dizer que qualquer problema pode ser codificado usando qualquer linguagem, afinal todo programa é de fato um conjunto de instruções dado em linguagem de máquina, mas será que isso seria conveniente?

As linguagens de programação podem ser classificadas de acordo com o grau de abstração que oferecem, ou seja, quão permissiva ela é para que o programador se concentre nos aspectos essenciais e ignore detalhes menos importantes ou minúcias operacionais enquanto cria seu código.

Deste modo, as linguagens que oferecem um alto grau de abstração, geralmente mais próximo da linguagem humana, permitindo descrever e relacionar os elementos de programação em um nível mais abstrato e geral, são chamadas de linguagem de alto nível. Por sua vez, aquelas mais próximas das características de máquina e de sua forma funcional, em que o conhecimento de detalhes relativos à arquitetura e à organização é fundamental, são ditas linguagens de baixo nível.

1.3 Identificação e diferenciação dos elementos do domínio

No contexto de compiladores é importante ter domínio do que os diferentes termos do domínio significam. Embora cumprindo tarefas parecidas, os compiladores, os tradutores, os montadores e os interpretadores são elementos diferentes.

Compiladores: traduzem programas codificados em linguagens de alto nível para linguagem de baixo nível. Exemplos: C –> .EXE, Java -> *Bytecode* ou C# -> MSIL (*Microsoft Intermediate Language*)

Tradutores: traduzem programas codificados em linguagens de alto nível para linguagem de alto nível. Exemplos: Português –> Inglês ou C# -> Java

Montadores: traduzem programas codificados em linguagens de baixo nível (simbólica) para linguagem de baixo nível. Exemplo: Assembly –> .EXE

Interpretadores: diferenciam-se dos demais por realizarem a tradução do código-fonte e também executá-lo. Exemplos: Browser -> HTML, Bash (Linux) -> Scripts

1.4 Ferramentas de apoio

Às vezes, certas funções são erroneamente atribuídas aos compiladores. O uso de um "ambiente de desenvolvimento" é bastante comum para ajudar nas diversas atividades que compreendem a construção de um programa. Juntamente com os compiladores, atua um conjunto de ferramentas que tem por objetivo automatizar tarefas e aumentar a produtividade. Dentre as mais famosas, podemos citar como exemplos:

Interfaces Integradas de Desenvolvimento (IDEs): costumeiramente, oferecem diversos recursos, mas, principalmente, auxiliam no gerenciamento de arquivos relativos ao projeto, permitem a execução e a depuração da aplicação sem que seja necessário deixar o ambiente de desenvolvimento, complemento de comandos e realce de sintaxe.

Depuradores (*debuggers*): permitem execução passo a passo do código, acompanhamento do valor de variáveis e dos conteúdos de memória durante a execução, a inserção de pontos específicos de parada, dentre outros.

Editores gráficos de interface: muito útil em projetos que tenham janelas e outros elementos gráficos, habitualmente permitem a inserção e o posicionamento de componentes usando o *mouse*.

1.5 A construção de compiladores: o modelo de Análise e Síntese

A tarefa de tradução realizada por um compilador pode ser bastante desafiadora, pois implica em analisar o código escrito pelo programador, compreendendo sua estrutura e suas relações entre os elementos e construir um código novo que seja equivalente, mas que utilize uma linguagem de programação diferente e com baixo nível de abstração.

Há aqui certa ironia e que nos faz parecer estar em uma espiral: a tarefa que precisa ser realizada pelo compilador se assemelha com aquela desempenhada pelo próprio programador quando cria o seu código, que realiza a tradução de uma ideia que pode ser bastante abstrata para uma linguagem de programação que lhe impõe certas restrições.

Para garantir o sucesso dessa empreitada convém dividir a tarefa, resolvendo os problemas por partes e de modo sucessivo. Embora haja diferentes estratégias para a construção de compiladores, praticamente todas elas seguem essa mesma abordagem.

O modelo de construção, conhecido como de Análise e Síntese, tornou-se o mais adotado e se fundamenta em dividir o processo em duas grandes fases: a análise, em que o código-fonte será verificado e um modelo representativo de sua estrutura e significados é construído; e a síntese, em que o código correspondente à tradução será efetivamente produzido de acordo.

A análise léxica (ou *scanner*) é a primeira etapa do processo de compilação e sua função é varrer o código-fonte, caractere por caractere, compondo os elementos que, de fato, formam o programa e que são chamados de *tokens* ou lexemas. Adicionalmente, elementos tais como espaços em branco, tabulações, marcas de formatação de texto, comentários ou qualquer outro que sejam irrelevantes ao processo de compilação são eliminados. Desse modo, dizemos que o analisador léxico é responsável por transformar o fluxo de caracteres em um fluxo de *tokens*.

Garantida a produção de uma sequência coerente de lexemas, precisamos ainda verificar a adequação da estrutura gramatical do programa. A próxima subfase é a análise sintática em que se busca determinar se uma sequência de símbolos léxicos (cadeia) pode ser gerada pela gramática que define a linguagem em questão. Essa tarefa consiste em construir a árvore de derivação correspondente.

Resta ainda verificar o significado ou a semântica do programa, tarefa executada pela última etapa da análise, a chamada Análise Semântica. É de sua responsabilidade verificar certos aspectos que não foram investigados anteriormente por impossibilidade ou inadequação de propósito, tais como questões relativas a escopo, regras de visibilidade e compatibilidade entre tipos.

Uma vez garantido que o código-fonte é válido e coerente, pode-se passar ao processo de síntese. É durante a geração de código que efetivamente ocorre a produção do código equivalente ao programa original. Dado o nível de dificuldade inerente à diferença entre os níveis de abstração, uma primeira tentativa pode ser realizada e seu produto passar por uma fase de otimização, cujo objetivo é melhorar o código gerado, eliminar redundâncias e aumentar o desempenho.

Leituras recomendadas:

Capítulo 1, principalmente as seções 1.1 a 1.4, do livro "Compiladores: Princípio, Técnicas e Ferramentas" (livro do Dragão)

Capítulo 1 completo do livro "Implementação de Linguagens de Programação: Compiladores"

Exercícios resolvidos:

1. Uma discussão típica entre desenvolvedores apaixonados é a de que a linguagem de programação de sua preferência é a melhor. Um exemplo desses embates ocorreu em certa ocasião em que um desenvolvedor Java e um programador C trocavam insultos a respeito da linguagem predileta do outro, dizendo:

– C é uma porcaria! Se quiser portar uma aplicação, vai ter que arrumar o código-fonte e recompilar ... isso se não der problema!

– Java é lento, pois é uma linguagem interpretada. Nunca se comparará com o desempenho de um código compilado!

a) Em que aspectos os comentários podem ser verdadeiros?

Resp.: Compiladores traduzem o código-fonte para uma linguagem alvo, específica de uma plataforma. Deste modo, se quiser mudar para uma plataforma diferente é necessário recompilar o código-fonte considerando a nova arquitetura. Nesse processo algumas bibliotecas específicas podem não ter sido portadas ou não haver uma versão similar que a substitua nessa nova plataforma, causando aborrecimentos ou até mesmo inviabilizando todo o processo.

A linguagem Java é um bom exemplo de técnica híbrida, pois os códigos-fontes escritos em Java são compilados, tendo como linguagem alvo os bytecodes da máquina virtual. Quando executados, os programas então interpretados pela máquina virtual e, dessa forma, realizam um processo de tradução dos bytecodes do programa para as instruções da plataforma utilizada naquele momento. É certo que o tempo adicional gasto com a interpretação não seria necessário se o programa fosse compilado diretamente para a plataforma de destino.

b) De que forma a linguagem preferida por cada um deles poderia ser considerada como melhor em relação à característica que o interlocutor apontava como uma deficiência na linguagem de seu desafeto?

Resp.: Um programa em Java sempre é executado na máquina virtual. A portabilidade se dá não pelo programa em si, mas pela máquina virtual que é quem interpreta os bytecodes nas instruções específicas daquela plataforma. Portanto, supostamente onde houver uma versão da máquina virtual implementada é possível executar aquele mesmo programa sem ter que compilá-lo novamente.

O fato de um programa ser compilado uma única vez permite que seja executado inúmeras vezes sem que nenhum outro processo de tradução aconteça. Mas se todas as vezes em que for executado tiver que passar por mais uma etapa, a de interpretação por uma máquina virtual por exemplo, é certo que consumirá algum tempo nessa tarefa adicional.

2. Conceitue cada um dos elementos dados a seguir:

a) Linguagem-fonte e objeto;

Resp.: Linguagem-fonte é a linguagem usada para escrever programas que são entrada de processadores de linguagens. Linguagem-objeto é a linguagem usada para escrever programas que são saída de processadores de linguagens.

b) Linguagem de alto nível e de baixo nível;

Resp.: Linguagem de alto nível é a linguagem mais próxima da linguagem natural e apresenta como principais características portabilidade, segurança, legibilidade e uso de abstrações. Linguagem de baixo nível é a linguagem mais próxima do hardware e apresenta como principais características dependência da arquitetura, baixa legibilidade, baixa segurança e pouco ou nenhum suporte para o uso de abstrações.

c) Linguagem de montagem e de máquina;

Resp.: Linguagem de montagem e linguagem de máquina são ambas linguagens de baixo nível e por isso compartilham praticamente as mesmas características. A única diferença é que a linguagem de montagem utiliza mnemônicos para melhorar um pouco a legibilidade dos programas, ao passo que a linguagem de máquina utiliza apenas os códigos numéricos que são interpretados pela máquina-alvo diretamente.

d) Discorra sobre as relações que existem entre todos esses tipos de linguagens.

Resp.: Linguagens-fonte e linguagens-objeto podem ser tanto de alto quanto de baixo nível. Conforme a particular combinação, dá-se o nome ao processador de linguagens como compilador, tradutor, filtro, montador etc. Linguagens de alto nível e de baixo nível podem ser tanto linguagens-fonte quanto linguagens-objeto, dependendo de serem, respectivamente, entrada ou saída de processadores de linguagens.

2 ANÁLISE LÉXICA

Antes de tentar converter o algoritmo em código de máquina, deve-se analisar o código-fonte para se assegurar que esteja correto, tanto em termos estruturais quanto semanticamente. Para isso é imprescindível identificar os elementos que compõem o código e reconhecê-los de maneira apropriada, inclusive diferenciando eventuais sequências similares.

A análise léxica

Cabe ao analisador léxico separar e identificar os elementos válidos que compõem o programa fonte, transformando um fluxo de caracteres em um fluxo de elementos significativos (vide figura 2). Por meio do processamento individual dos caracteres do arquivo de entrada, os símbolos são agrupados segundo sua representatividade na linguagem fonte, formando um elemento atômico chamado *token*.

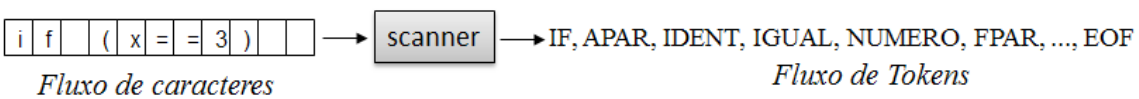

Figura 2 - Transformação de fluxo de caracteres para um fluxo de tokens

Isso significa que, concluída a análise léxica, cada um dos elementos utilizados para a escrita do programa, tais como identificadores, operadores, delimitadores e palavras reservadas terão sido reconhecidos e identificados.

Compete, ainda, ao analisador léxico a tarefa de eliminar os chamados "elementos decorativos" do programa, isto é, caracteres que não possuem relevância funcional dentro do código, tais como espaços em branco excedentes, marcas de formatação de texto, tabulações e comentários.

Assim, as seguintes tarefas que são pertinentes:

- Esquadrinhar o código-fonte, símbolo a símbolo, compondo *tokens* e classificando-os (segundo seu significado para a linguagem);
- Compor e gerenciar a chamada "Lista de Tokens", uma relação de todos os elementos identificados pelo *scanning* em uma lista linear;
- Eliminar elementos desnecessários ao processo de compilação, tais como comentários e símbolos decorativos;
- Reconhecer e validar sequências numéricas, quer sejam inteiros, reais ou outra base suportada pela linguagem (e.g. hexadecimal);
- Reconhecer e validar elementos de definição pelo programador e utilizados como identificadores;
- Prover um mecanismo para controle de erros amigável, haja vista que é o analisador léxico quem varre o código-fonte e, portanto, é o único que tem referência da localidade em que um determinado *token* ocorre (e.g. mensagem de erro relativa ao elemento E da linha N).

2.1 Como identificar e reconhecer os *tokens* em meio ao texto?

Os *tokens* possuem uma estrutura sintática e, desse modo, podemos descrever cada um dos itens da linguagem (palavras reservadas, operadores, delimitadores etc.) por meio de regras de produção, tais como:

identif = letra {letra|dígito}
número = dígito {dígito}
if = "i""f"
igual = "=""="

Pensando nisso, inevitavelmente surge a pergunta: *Por que o analisador léxico não é uma parte do próprio analisador sintático*?

Observe alguns aspectos que justificam essa divisão de fases:

1. Isso deixaria o analisador sintático mais complicado de ser construído, a citar a dificuldade para distinguir entre palavras reservadas e identificadores. Por exemplo, uma regra sintática que poderia ser escrita assim:

Statement ::= ident "=" Expr ";" |
"if" "(" Expr ")" ...

Precisaria ser reescrita desta forma para que fosse possível tratar as duas tarefas simultaneamente:

Statement ::= "i" ("f" "(" Expr ")" ... | notF {letter | digit} "="
Expr ";") | notI {letter | digit} "=" Expr ";"

2. O *scanning* deve eliminar brancos, tabulações, fins de linha e comentários. Considerando que esses caracteres podem ocorrer em qualquer lugar do código, teríamos que especificar gramáticas mais complexas. Exemplo:

Statement ::= "if" {Blank} "(" {Blank} Expr {Blank} ")" {Blank} ...
Blank ::= " " | "\r" | "\n" | "\t" | Comment

A estrutura dos *tokens* pode ser descrita por gramáticas regulares, que são mais simples e mais eficientes que as gramáticas livres de contexto. A maioria das estruturas léxicas é regular:

Nomes ::= letra { letra | dígito }
Números ::= dígito { dígito }
Strings ::= "\"" { qqCaractereExcetoAspas } "\""
Palavras reservadas ::= letra { letra }
Operadores ::= ">" | "=" | "+" | ...

Gramáticas regulares não podem lidar com estruturas aninhadas, por não serem capazes de manipular recursão central. Esse tipo de construção é importante na maioria das linguagens de programação.

- Expressões aninhadas:

Expr → ... "(" Expr ")" ...

- Comandos aninhados:

Comando → "do" Comando "while" "(" Expr ")"

- Classes aninhadas:

Classe → "class" "{" ... Classe ... "}"

Os autômatos finitos são reconhecedores de linguagens regulares. Assim, podemos definir o *scanner* como sendo um AFD (Autômato Finito Determinístico) que reconheça os elementos da linguagem fonte. O processamento de cada caractere obtido será regido pelas transições previstas pelo autômato, de modo que o AFD reconheça uma sentença se:

- A entrada tiver sido consumida totalmente ou

- Não for possível realizar uma transição com o próximo símbolo da entrada e o autômato se encontrar em um estado final.

Veja um exemplo no esquema representado pela figura a seguir (figura 3):

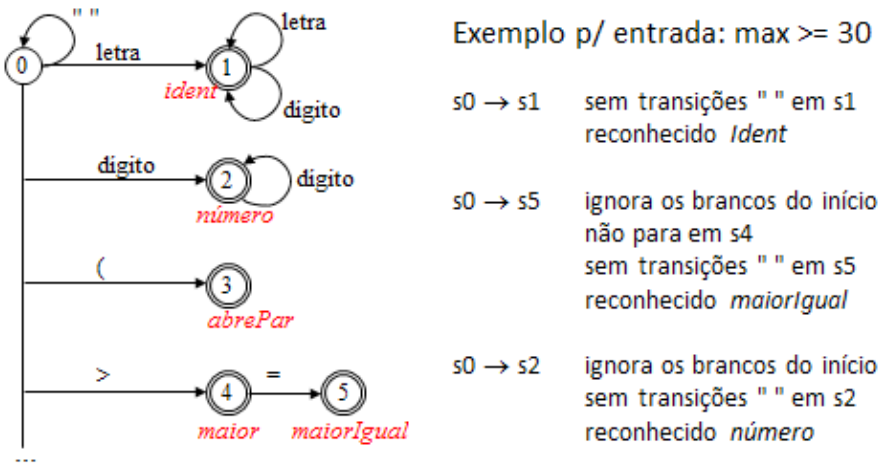

Figura 3 - Ilustração do processo de reconhecimento de lexemas por um autômato finito

Leituras recomendadas:

Capítulo 1, principalmente as seções 1.1 a 1.4, do livro "Compiladores: Princípio, Técnicas e Ferramentas" (livro do Dragão)

Capítulo 1 completo do livro "Implementação de Linguagens de Programação: Compiladores"

Exercício resolvido:

1. Em meio a um código-fonte, há elementos considerados desnecessários ao processo de compilação, tais como espaços em branco e quebras de linha, dentre outros. Se os espaços em branco são necessários para que se possa separar uma palavra de outra, então de que modo são considerados desnecessários?

Resp.: Apenas um caractere de espaço é necessário para separar uma palavra de outra, então ocorrências consecutivas são desnecessárias para esse propósito. Mas até mesmo esse único espaço perde sua utilidade após a identificação do término do token em formação e, consequentemente, também pode ser descartado após cumprir com o seu propósito.

3 ANÁLISE SINTÁTICA (PARTE 1)

Tendo a função de verificar se as construções utilizadas no código-fonte estão na forma apropriada, a análise sintática tem a responsabilidade de:

Dada uma gramática livre de contexto G e uma sentença s, aqui representada pelo programa fonte, o analisador sintático tem o propósito de verificar se s pertence à linguagem gerada por G. Em outras palavras, a partir dos *tokens* fornecidos pelo analisador léxico, tentar construir a árvore de derivação para s segundo as regras de produção dadas pela gramática G. Se essa tarefa for possível, o programa é considerado sintaticamente correto.

3.1 Tarefas que são pertinentes

Comprovar que a sequência de *tokens* cumpre as regras gramaticais e gerar a árvore gramatical do programa.

Assim, entendemos como sendo de sua competência:

- A identificação de erros de sintaxe

  – Por exemplo: A * / B

- A correta interpretação da estrutura hierárquica e a evolução da sentença, mesmo quando sob aspectos implícitos.

  – Por exemplo, a sentença A / B * C será entendida como (A/B) * C em Fortran e A / (B*C) em APL

- A capacidade de tratamento e/ou recuperação de erros de sintaxe. Vale ressaltar que em relação a esse aspecto, o projeto oferece recursos suficientes, mas também tem o cuidado de não retardar de forma significativa o processamento de programas corretos.

Há claras vantagens na utilização de gramáticas para especificar linguagens de programação e projetar compiladores, a citar:

- Especificações sintáticas precisas de linguagens;
- Possibilidade de uso de ferramentas para a geração automática do parser;
- Possibilidade de identificar ambiguidades durante o processo de especificação/construção;
- Facilidades para ampliar ou modificar a linguagem.

3.2 Tratamento / recuperação de erros

Em geral, programas contêm erros. Assim, um bom compilador deve, na medida do possível, detectar todos os erros ocorridos, avisar o usuário de suas ocorrências e recuperar-se deles de modo que seja possível analisar o restante do código-fonte.

A implementação de um bom mecanismo de detecção e recuperação de erros muitas vezes depende de se essa questão foi considerada desde o início do projeto da linguagem.

Modo pânico ou desespero: Para imediatamente diante do primeiro erro ou, identificado um erro, o analisador sintático descarta símbolos de entrada, até que seja encontrado um *token* pertencente ao subconjunto de *tokens* de sincronização (e.g. ponto e vírgula, delimitadores etc.).

Recuperação de frases: Ao descobrir um erro, o analisador sintático pode tentar uma recuperação realizando uma correção local na entrada restante, substituindo-a por alguma cadeia que permita que a análise prossiga. Por exemplo, substituir uma vírgula inadequada por um ponto e vírgula, ou ainda remover um ":" excedente.

Produções de erro: Modificar a gramática incluindo regras de produção estratégicas de modo a acomodar os erros mais comuns. Assim, sempre que uma produção de erro for identificada pelo analisador é possível conduzir o tratamento mais adequado àquela situação em especial.

Correção global: Em geral, emprega algoritmos que escolhem entre possíveis soluções, aquela que apresenta uma sequência mínima de mudanças para se obtenha uma correção global do programa e não dos erros individualmente.

Alguns dados interessantes que devem ser considerados quando se pensa em quão eficiente deve ser o método de tratamento e recuperação de erros *versus* o tempo demandado para que opere de modo satisfatório:

Cerca de 60% dos programas compilados corretos sintaticamente e semanticamente. Dentre os que apresentam erros, cerca 80% dos enunciados apresentam apenas um erro e 13% apresentam apenas dois erros. Em, aproximadamente, 90% dos casos, os erros envolvem um único *token*.

3.3 Análise sintática descendente e ascendente

Os métodos de análise sintática podem ser classificados segundo a maneira pela qual a árvore de derivação da cadeia analisada x é construída:

- Nos métodos descendentes, a árvore de derivação correspondente a x é construída de cima para baixo, ou seja, da raiz (o símbolo inicial S) para as folhas, onde se encontra x.

- Nos métodos ascendentes, a árvore de derivação correspondente a x é construída de baixo para cima, ou seja, das folhas, onde se encontra x, para a raiz, onde se encontra o símbolo inicial S.

Nos métodos descendentes ou *top-down*, temos de decidir qual a regra $A \rightarrow \alpha$ a ser aplicada a um nó rotulado por um não terminal A. A expansão de A é feita criando nós filhos rotulados com os símbolos de α.

Nos métodos ascendentes ou *bottom-up*, temos de decidir quando a regra $A \rightarrow \alpha$ deve ser aplicada e devemos encontrar nós vizinhos rotulados com os símbolos de α. A redução pela regra $A \rightarrow \alpha$ consiste em acrescentar à árvore um nó A, cujos filhos são os nós correspondentes aos símbolos de α.

Métodos descendentes e ascendentes constroem a árvore da esquerda para a direita. A razão para isso é que as escolhas das regras devem se basear na cadeia a ser gerada, que é lida da esquerda para a direita. (Seria muito estranho um compilador que começasse a partir do fim do programa em direção ao início)

Por exemplo, considere a cadeia x = a + a * a e a gramática:

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

$$P = \{ \quad 1.\ E \rightarrow E + T \quad ,$$
$$2.\ E \rightarrow T \quad ,$$
$$3.\ T \rightarrow T * F \quad ,$$
$$4.\ T \rightarrow F \quad ,$$
$$5.\ F \rightarrow (E) \quad ,$$
$$6.\ F \rightarrow a \quad \}$$

Usando-se um método descendente, a árvore de derivação de x é construída na sequência especificada conforme dado na figura a seguir:

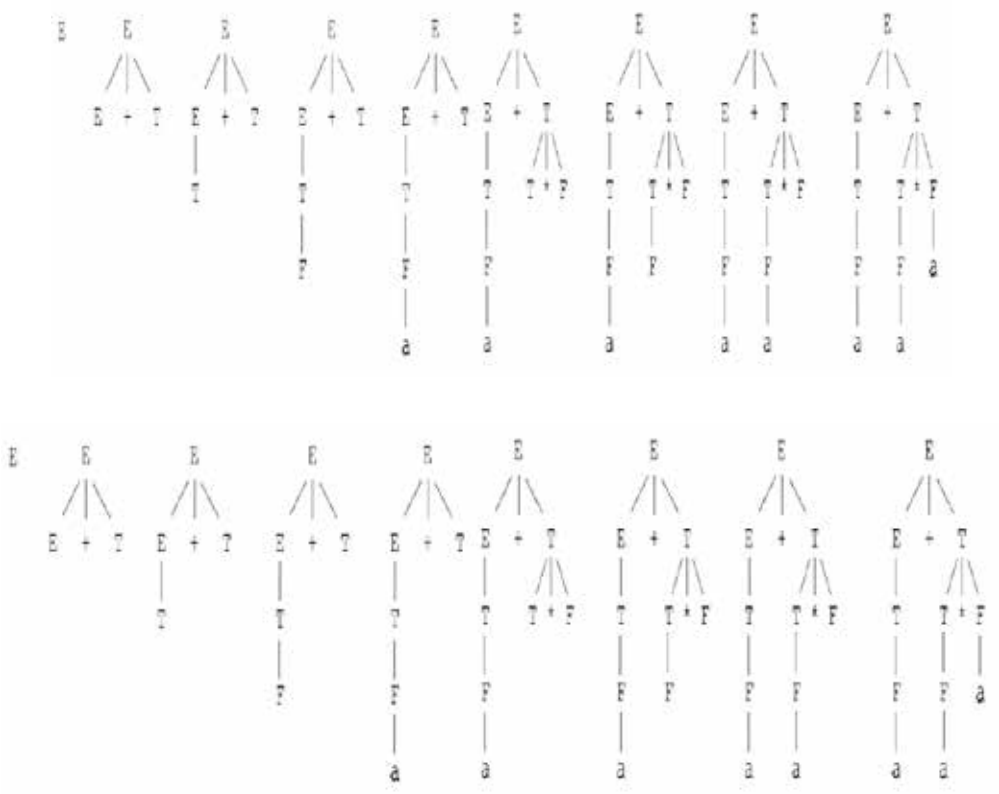

Figura 4 - Etapas da construção da árvore sintática da sentença a+a*a por um método descendente

Note que as regras são consideradas na ordem 1 2 4 6 3 4 6 6, a mesma ordem em que as regras são usadas na derivação esquerda: $E \Rightarrow E+T \Rightarrow T+T \Rightarrow a+T \Rightarrow a+T*F \Rightarrow a+F*F \Rightarrow a+a*F \Rightarrow a+a*a$

Porém, se usarmos um método de análise ascendente, as regras são identificadas na ordem 6 4 2 6 4 6 3 1 e os passos de construção da árvore podem ser vistos na figura a seguir:

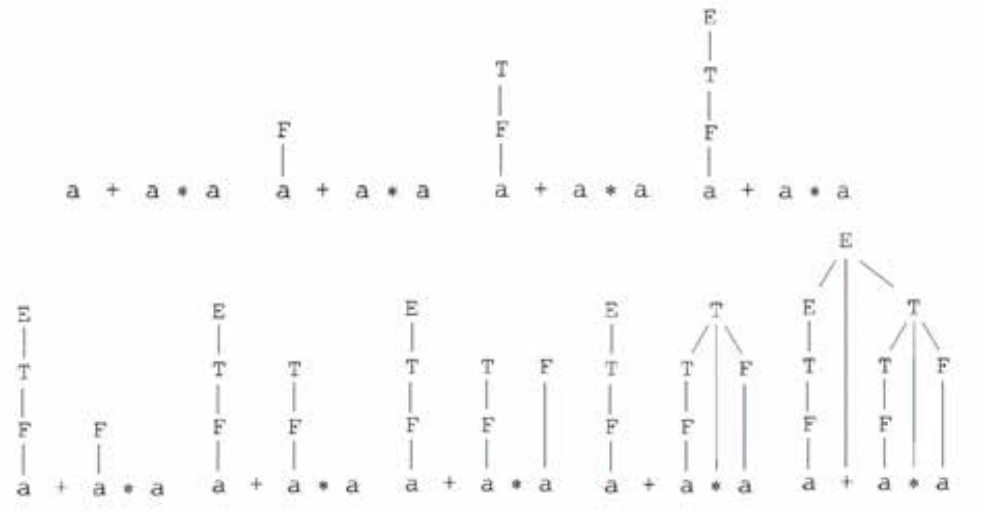

Figura 5 - Etapas da construção da árvore sintática da sentença a+a*a por um método ascendente

Embora a árvore de derivação seja usada para descrever os métodos de análise, na prática, ela nunca é efetivamente construída. Às vezes, se necessário, construímos "árvores sintáticas", que guardam alguma semelhança com a árvore de derivação, mas ocupam um espaço de memória significativamente menor. A única estrutura de dados necessária para o processo de análise é uma pilha, que guarda informação sobre os nós da árvore de derivação relevantes, em cada fase do processo.

No caso da análise descendente, os nós relevantes são aqueles ainda não expandidos; no caso da análise ascendente, são as raízes das árvores que ainda não foram reunidas em árvores maiores.

3.4 Tipos de analisadores sintáticos

Além de categorizá-los quando ao percurso de construção da árvore sintática, os métodos de análise sintática ainda podem ser classificados segundo a forma como operam; isto é, quantos elementos da entrada são necessários para se decidir qual regra deve ser aplicada, se apresentam uma tabela de movimento e como ela é montada etc.

A seguir, citamos alguns exemplos de analisadores segundo seu tipo:

- Métodos descendentes (*Top-Down*): constroem a árvore sintática de cima para baixo
  - Método de *Earley*
  - Analisadores descendentes recursivos
  - Analisadores LL(k)

- Métodos ascendentes (*Bottom-up*): constroem a árvore sintática de baixo para cima
  - *Cocke-Younger-Kasami*
  - Analisadores SR
  - Analisadores LR
  - Analisadores LALR

Vale destacar que os métodos de *Cocke-Younger-Kasami* e *Earley* são tidos como universais e servem para qualquer gramática, podendo ser considerados ainda bem eficientes.

Leituras recomendadas:

Capítulo 4, seções 4.1 e 4.2, do livro "Compiladores: Princípio, Técnicas e Ferramentas" (livro do Dragão)

Capítulo 3, seção 3.1, do livro "Implementação de Linguagens de Programação: Compiladores"

Exercícios resolvidos:

1. Conceitue cada um dos itens dados a seguir:

Análise sintática determinística x não determinística;

Análise sintática descendente x ascendente;

Gramática e linguagem LL(1) e LR(1).

Resp.: Análise determinística é aquela em que todas as decisões de movimentação do reconhecedor são tomadas "sem arrependimento". Se não houver movimentação possível, isso indica a ocorrência de um erro na cadeia de entrada. Análise não determinística, por outro lado, opera por tentativa e erro. Erros na cadeia de entrada são apontados apenas depois de esgotadas todas as possibilidades de movimentação.

A análise descendente, também conhecida como top-down, é aquela em que o reconhecedor inicia os seus movimentos na raiz da gramática e evolui até a cadeia de entrada (em termos de árvore, é aquela em que a árvore é montada de cima para baixo). Análise ascendente, ou bottom-up, é aquela em que os movimentos do reconhecedor vão da cadeia de entrada em direção à raiz da gramática (a árvore é montada de baixo para cima).

Uma Gramática LL(1) é aquela que gera uma linguagem que pode ser reconhecida da esquerda para a direita, usando derivações mais à esquerda e com lookahead de apenas um símbolo. Linguagem LL(1) é aquela que pode ser gerada por alguma gramática LL(1). Gramática LR(1) é aquela que gera uma linguagem que pode ser reconhecida da esquerda para a direita, usando

reduções mais à esquerda e com lookahead de apenas um símbolo. Linguagem LR(1) é aquela que pode ser gerada por alguma gramática LR(1).

2. Considere a seguinte gramática:

G = ({L, S, I}, {"a", ",", "(", ")"}, P, L)
P = { 1. L → "(" S ")" ,
2. S → I "," S | I ,
3. I → "a" | L }

a) Mostre os movimentos de um reconhecedor ascendente na análise da sentença (a,(a),(a,a));

b) Mostre os movimentos de um reconhecedor descendente na análise da sentença (a,(a),(a,a));

c) Obtenha o esboço de um reconhecedor recursivo descendente para a linguagem por ela definida.

Resp. item a: Os movimentos de um reconhecedor ascendente serão

(a,(a),(a,a)) ⇒ (I,(a),(a,a)) ⇒ (I,(I),(a,a)) ⇒ (I,(S),(a,a)) ⇒ (I,L,(a,a)) ⇒ (I,I,(a,a)) ⇒ (I,I,(I,a)) ⇒ (I,I,(I,I)) ⇒ (I,I,(I,S)) ⇒ (I,I,(S)) ⇒ (I,I,L) ⇒ (I,I,I) ⇒ (I,I,S) ⇒ (I,S) ⇒ (S) ⇒ L

Resp. item b: Os movimentos para o reconhecedor descendente serão

L ⇒ (S) ⇒ (I,S) ⇒ (a,S) ⇒ (a,I,S) ⇒ (a,L,S) ⇒ (a,(S),S) ⇒ (a,(I),S) ⇒ (a,(a),S) ⇒ (a,(a),I) ⇒ (a,(a),L) ⇒ (a,(a),(S)) ⇒ (a,(a),(I,S)) ⇒ (a,(a),(a,S)) ⇒ (a,(a),(a,I)) ⇒ (a,(a),(a,a))

Resp. item c: Um reconhecedor recursivo é obtido transcrevendo cada uma das regras de produção da gramática como uma sub-rotina responsável por consumir os tokens da cadeia gerada. Sempre que encontrarmos um símbolo não terminal na sentença, realiza-se a chamada sub-rotina correspondente e, quando encontrarmos um símbolo terminal na regra, verificamos se corresponde ao símbolo dado na entrada (posição corrente de análise de código-fonte). De acordo com esses conceitos, poderíamos ter o seguinte código para essa gramática:

```
void parseL() {
 accept ("(");
 parseS();
 accept (")");
}
```

```
void parseS() {
 parseI();
 while s==",," {
 acceptIt();
 parseI();
 }
}
```

```
void parseI() {
 switch s {
 case "a": acceptIt();
 break();
 case "(": parseS();
 break;
 default: ERRO();
 }
}
```

4 ANÁLISE SINTÁTICA (PARTE 2 – DESCENDENTE)

A representação do processo será feita por meio de configurações (a, y), em que a e y representam, respectivamente, o conteúdo da pilha e o resto da entrada ainda não analisada. Por convenção vamos supor que o topo da pilha fica à esquerda, isto é, que o primeiro símbolo de a é o símbolo do topo da pilha.

Existem duas formas de transição de uma configuração para outra:

(1) **expansão** de um não terminal pela regra $A \rightarrow \alpha$: permite passar da configuração $(A\beta, y)$ para a configuração $(\alpha\beta, y)$.

(2) **verificação** de um terminal s: permite passar da configuração $(s\alpha, sy)$ para a configuração (α, y).

O segundo tipo de transição serve para retirar terminais do topo da pilha e expor no topo da pilha o próximo não terminal a ser expandido. A configuração inicial para a entrada x é (S, x); o processo termina na configuração $(\varepsilon, \varepsilon)$, ou seja, com a pilha vazia e a entrada toda considerada.

Mostramos abaixo as configurações sucessivas de um analisador descendente, para a cadeia x. Para acompanhamento, a terceira coluna apresenta os passos correspondentes da derivação esquerda de x. Note que apenas os passos de expansão têm correspondente na derivação.

Pilha (topo à esquerda)	(Restante da) entrada	Derivação esquerda (left-most)
E	a+a*a	E
E+T	a+a*a	⇒ E+T
T+T	a+a*a	⇒ T+T
F+T	a+a*a	⇒ F+T
a+T	a+a*a	⇒ a+T
+T	+a*a	
T	a*a	
T*F	a*a	⇒ a+T*F
F*F	a*a	⇒ a+F*F
A*F	a*a	⇒ a+a*F
*F	*a	
F	a	
a	a	⇒ a+a*a
ε	ε	

Figura 6 - Correspondência entre as operações de um parser descendente e do processo de derivação do não terminal mais à esquerda (left-most) durante a validação da cadeia a+a*a

4.1 Análise sintática LL(1)

Neste método, a escolha da regra a ser usada durante o processo de análise descendente se dá por meio de duas informações: o não terminal A a ser expandido e o primeiro símbolo a do resto da entrada.

Uma tabela M com essas informações indexadas por essas duas entradas nos fornece a regra a ser utilizada: M[A, a]. Vale ressaltar que essa técnica só pode ser empregada para uma classe restrita de gramáticas, as chamadas gramáticas LL(1).

O nome LL(1) indica que:

- A cadeia de entrada é examinada da esquerda para a direita (L = *left-to-right*);
- O analisador procura construir uma derivação esquerda (L = *leftmost*);
- Exatamente 1 símbolo do resto da entrada é examinado.

Exemplo: Suponha a gramática a seguir:

$$G = (\{E, T, F, E', T', F'\}, \{a, +, *, (,)\}, P, E)$$

$$P = \{ \quad 1.\ E \rightarrow T\,E' \quad ,$$
$$2.\ T \rightarrow F\,T' \quad ,$$
$$3.\ F \rightarrow (\,E\,) \quad ,$$
$$4.\ F \rightarrow a \quad ,$$
$$5.\ E' \rightarrow +\,T\,E' \quad ,$$
$$6.\ E' \rightarrow \varepsilon \quad ,$$
$$7.\ T' \rightarrow *\,F\,T' \quad ,$$
$$8.\ T' \rightarrow \varepsilon \quad \}$$

Essa gramática é LL(1) e, assim, a sua tabela de análise M será por:

	(	a	+	*	)	$
E	1	1	-	-	-	-
T	2	2	-	-	-	-
F	3	4	-	-	-	-
E'	-	-	5	-	6	6
T'	-	-	8	7	8	8

Figura 7 - Tabela de movimentos para o analisador LL(1)

Nessa tabela (figura 7), a entrada M[*S*, *k*] correspondente ao não terminal *S* e ao terminal *k* contém o número da regra que deve ser usada para expansão de *S*. As entradas sinalizadas por "-" correspondem

a combinações decorrentes de erros, ou seja, são casos impossíveis de ocorrer durante a construção de uma sentença válida para a linguagem.

Para analisar a cadeia a + a * a, teremos as seguintes configurações:

Pilha (topo à esquerda)	(Restante da) entrada	Escolha da regra
E	a + a * a	M[E, a] = 1
T E'	a + a * a	M[T, a] = 2
F T' E'	a + a * a	M[F, a] = 4
a T' E'	a + a * a	< *Verificação* >
T' E'	+ a * a	M[T', +] = 8
E'	+ a * a	M[E', +] = 5
+ T E'	+ a * a	< *Verificação* >
T E'	a * a	M[T, a] = 2
F T' E'	a * a	M[F, a] = 4
a T' E'	a * a	< *Verificação* >
T' E'	* a	M[T', *] = 7
* F T' E'	* a	< *Verificação* >
F T' E'	A	M[F, a] = 4
a T' E'	A	< *Verificação* >
T' E'	ε	M[T', $] = 8
E'	ε	M[E', $] = 6
ε	ε	< *Verificação* >

Figura 8 - Análise da sentença a+a*a pelo parser LL(1), segundo o que é dado pela tabela de movimentos

Vamos agora mostrar como construir M, a tabela de análise LL(1). No caso mais simples, o símbolo a (o primeiro do resto da entrada) é o primeiro símbolo derivado do não terminal a ser expandido A e faz parte de First(A). Nesse caso, a deve pertencer a First(α), em que $A \rightarrow \alpha$ é uma das alternativas de regra para A. Como ilustração, podemos ver que, no exemplo acima, a regra $F \rightarrow (E)$ foi usada com o símbolo (.

Outra possibilidade é a de que não seja A o não terminal responsável pela geração do símbolo a, mas sim algum outro não terminal encontrado depois de A. Nesse caso, devemos ter a pertencendo ao Follow(A). Como ilustração, podemos ver que, no exemplo acima, a regra $T' \rightarrow \varepsilon$ foi usada com o símbolo +.

Para construir a tabela M, vamos examinar todas as regras da gramática:

- Para cada regra i: $A \rightarrow \alpha$, temos M[A, a] = i, para cada a em First(α).
- Para cada regra i: $A \rightarrow \alpha$, se $\alpha \Rightarrow^* \varepsilon$, temos M[A, a] = i, para cada a em Follow(A).

Se a gramática é LL(1), cada entrada de M receberá, no máximo, um valor. As entradas de M que não receberem nenhum valor devem ser marcadas como entradas de erro. Caso alguma entrada de M seja definida mais de uma vez, dizemos que houve um conflito e que a gramática não é LL(1).

Considere a gramática dada no exemplo anterior. Essa gramática é LL(1) e a tabela de análise M pode ser construída como indicado na figura 9. Teremos, então, para cada regra:

Regra	Conjunto a ser considerado	Entradas da tabela a serem marcadas
1. E → TE'	First(TE') = { (, a }	M[E, (] = 1 e M[E, a] = 1
2. T → FT'	First(FT') = { (, a }	M[T, (] = 2 e M[T, a] = 2
3. F → (E)	First((E)) = { (}	M[F, (] = 3
4. F → a	First(a) = { a }	M[F, a] = 4
5. E' → +TE'	First(+TE') = { + }	M[E', +] = 5
6. E' → ε	Follow(E') = { $,) }	M[E', $] = 6 e M[E',)] = 6
7. T' → *FT'	First(*FT') = { *}	M[T',*] = 7
8. T' → ε	Follow(T') = { $, +,)}	M[T', $] = 8, M[T', +] = 8 e M[T',)] = 8

Figura 9 - Determinando as entradas para a tabela de movimentos do Parser LL(1)

Considere agora outra gramática, que é equivalente a dada no exemplo anterior.

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

$$P = \{ \quad 1.\ E \rightarrow E + T \quad , \quad 2.\ E \rightarrow T \quad , \quad 3.\ T \rightarrow T * F \quad , \quad 4.\ T \rightarrow F \quad , \quad 5.\ F \rightarrow (E) \quad , \quad 6.\ F \rightarrow a \quad \}$$

Se analisarmos suas regras quanto aos elementos inicializadores (conjunto *First*), sucessores (conjunto *Follow*) e tentarmos montar a tabela de movimentos para o parser LL(1), teremos:

Regra	Conjunto a ser considerado	Entradas da tabela a serem marcadas
1. E → E+T	First(E+T) = { (, a }	M[E, (] = 1 e M[E, a] = 1
2. E → T	First(T) = { (, a }	M[E, (] = 2 e M[E, a] = 2
3. T → T*F	First(T*F) = { (, a }	M[T, (] = 3 e M[T, a] = 3
4. T → F	First(F) = { (, a }	M[T, (] = 4 e M[T, a] = 4
5. F → (E)	First((E)) = { (}	M[F, (] = 5
6. F → a	First(a) = { a }	M[F, a] = 6

Consequentemente, a gramática não é LL(1), por causa dos conflitos (múltiplas definições) para M[E, (], M[E, a], M[T, (] e M[T, a]. Em alguns casos, como o da gramática do exemplo anterior, é possível

concluir que a gramática não é LL(1) por inspeção. As duas características mais óbvias são: a recursão à esquerda e a possibilidade de fatoração.

Recursão à esquerda:

Se uma gramática permite uma derivação $A \Rightarrow^* A\alpha$, para algum não terminal A e para alguma cadeia não vazia α, a gramática é dita recursiva à esquerda.

No caso mais simples, existe na gramática uma regra $A \rightarrow A\alpha$ responsável diretamente pela derivação mencionada. Para que A não seja um não terminal inútil, deve existir na gramática (pelo menos) uma regra da forma $A \rightarrow \beta$, sem recursão à esquerda.

A combinação dessas duas regras faz com que First(A) e, portanto, First(Aα) contenham todos os símbolos de First(β) e isso leva necessariamente a um conflito. A eliminação da recursão à esquerda pode ser tentada, procurando transformar a gramática em uma gramática LL(1).

Observe que a combinação:

$A \rightarrow A\alpha$
$A \rightarrow \beta$

... permite a geração de cadeias da forma $\beta\alpha\alpha...\alpha\alpha\alpha\alpha$ e que essas mesmas cadeias podem ser geradas de outra maneira. Por exemplo:

$A \rightarrow \beta A'$
$A' \rightarrow \alpha A'$
$A' \rightarrow \varepsilon$

Essa outra forma de geração usa recursão à direita, que não cria problemas para os analisadores LL(1).

4.2 Prefixos comuns

Trata-se do caso em que duas regras começam com o mesmo símbolo ou conjunto de símbolos (prefixo), isto é, regras como $A \rightarrow \alpha\beta$ e $A \rightarrow \alpha\delta$, com $First(\alpha) \neq \varnothing$. Nesse caso, existe uma interseção entre First($\alpha\beta$) e First($\alpha\delta$), e a gramática não pode ser LL(1).

Isso acontece porque não é possível decidir, olhando apenas o primeiro símbolo derivado de α, qual a regra correta. A solução é simples e envolve a fatoração.

Suponha as regras abaixo:

$A \rightarrow \alpha\beta$
$A \rightarrow \alpha\delta$

Podemos reescrever essas regras em equivalentes de modo a colocar o prefixo em uma regra e adiar a decisão entre β e δ para quando o primeiro símbolo derivado de β ou de δ estiver visível.

$A \rightarrow \alpha A'$
$A' \rightarrow \beta$
$A' \rightarrow \delta$

Essas duas técnicas permitem transformar algumas gramáticas em gramáticas LL(1). Entretanto, vale ressaltar que algumas gramáticas livres de contexto GLC não têm gramáticas equivalentes LL(1). Nesse caso, a aplicação dessas técnicas ou mesmo de outras poderá não ter sucesso.

Vejamos um exemplo de como solucionar o problema de recursão à esquerda fazendo pequenas alterações nas regras da gramática de modo a transformá-las em recursões à direita. Considere a gramática a seguir:

$G = (\{E, T, F\}, \{a, +, *, (,)\}, P, E)$
$P = \{$ 1. $E \rightarrow E + T \mid T$,
2. $T \rightarrow T * F \mid F$,
3. $F \rightarrow (E) \mid a$,
4. $F \rightarrow a$ $\}$

Podemos notar que essa gramática apresenta duas situações de recursão à esquerda. Um caso associado ao símbolo E na regra 1 e o outro envolvendo o símbolo T, por meio da regra 2. Fazendo as substituições indicadas no processo descrito anteriormente, passaríamos a ter:

$G = (\{E, T, F, E', T'\}, \{a, +, *, (,)\}, P, E)$
$P = \{$ 1. $E \rightarrow T E'$,
2. $E' \rightarrow + T E' \mid \varepsilon$,
3. $T \rightarrow F T'$,
4. $T \rightarrow F * T' \mid \varepsilon$,
5. $F \rightarrow (E) \mid a$ $\}$

... e essa gramática é LL(1), como já foi visto anteriormente.

A eliminação de prefixos comuns também pode ser obtida por meio da reescrita de algumas regras da gramática. Suponha que tenhamos que transformar a gramática abaixo em uma gramática LL(1) equivalente.

$G = (\{L, S\}, \{if, then, else, fi, ;, exp\}, P, L)$
$P = \{$ 1. $L \rightarrow L ; S \mid S$,
2. $S \rightarrow if\ E\ then\ L\ else\ L\ fi \mid if\ E\ then\ L\ fi \mid S$,
3. $E \rightarrow exp$ $\}$

Observe que temos o problema de recursão à esquerda nas regras de L e também a possibilidade de fatoração nas regras de S. Uma transformação dessa gramática em LL(1) teria o seguinte resultado:

G = ({L, S, L', S'}, {if, then, else, fi, ;, exp}, P, L)
P = {
1. L → S L' | S ,
2. L' → ; S L' | ε ,
3. S → if E then L S' | S ,
4. S' → else L fi | fi ,
5. E → exp }

Novamente podemos verificar que essa é uma gramática LL(1), montando a tabela de movimentos para o parser LL(1) e observando que para nenhuma combinação de valores haverá duas ou mais regras que possam ser aplicadas simultaneamente.

Leituras recomendadas:

Capítulo 4, seção 4.4, do livro "Compiladores: Princípio, Técnicas e Ferramentas" (livro do Dragão)

Capítulo 3, seção 3.2, do livro "Implementação de Linguagens de Programação: Compiladores"

Exercícios resolvidos:

1. Considere a gramática abaixo:

G = ({S, X, Y, Z}, {a, b, c, d}, P, S)
P = {
1. S → aXb | aYc | aaZd ,
2. X → bX | bc ,
3. Y → cY | cb ,
4. Z → dZ | ε }

a) Essa gramática é LL(1)? Prove a sua resposta.

b) Caso a gramática acima não seja LL(1), obtenha uma gramática equivalente que seja LL(1) e prove que a nova gramática é de fato LL(1).

Resp. item a: Não, pois os conjuntos First não são disjuntos. Em todas as regras encontramos prefixos iguais para as possíveis derivações de cada regra, quer maneira direta ou indireta. Note que isso se reflete diretamente nos valores incluídos nos conjuntos First e Follow, conforme dados abaixo:

First (aXb) = {a}, First (aYc) = {a}, First (aaZd) = {a}

First (bX) = {b}, First (bc) = {b}

First (cY) = {c}, First (cb) = {c}

First (dZ) = {d}, follow (Z) = {d}

*Resp. item b: Eliminando recursões e fazendo substituições dos símbolos nas construções dadas por S, teremos: S → ab*bcb | ac*cbc | aad*d. Fatorando as três diferentes possibilidades de derivação para o símbolo S, ficaríamos com a seguinte regra: S → a(b*bcb | c*cbc | ad*d)*

*É possível observar que essa gramática não apresenta recursões à esquerda, uma vez que deriva diretamente para sentenças compostas apenas de símbolos terminais. Além disso, também nota-se que os prefixos para cada uma das possíveis derivações após o primeiro símbolo a são diferentes para os três casos possíveis. Esses aspectos podem ser confirmados quando calculamos os conjutos First para cada uma das três possibilidades de derivação apresentadas para o símbolo S: First (b*bcb) = {b}; First (c*cbc) = {c} e First (aad*d) = {a}.*

2. Obtenha o esboço de um reconhecedor, por meio do método recursivo descendente, para a linguagem definida pela expressão: (+|-|ε)(d+(ε|.|.d*)|.d+)(e(+|-|ε)d+|ε). São exemplos de sentenças pertencentes à essa linguagem: 123, -45.312, +.76, 5.44e2, +0.88e-35.

Resp.: Um reconhecedor recursivo transcreve a regra de produção como uma sub-rotina responsável por consumir dada um de seus tokens na ordem em que são dados. Sempre que encontrarmos um símbolo que pode ou não ocorrer na sentença, devemos ter o cuidado para não reportar um erro indevidamente. Assim, para a expressão dada, poderíamos ter o seguinte código:

```
void parseNumero() {
 if s=="+" takeIt();
 else if s=="-" takeIt();
 if s=="d" {
 takeIt();
 while s=="d" takeIt();
 if s=="." {
 takeIt();
 while s=="d" takeIt();
 }
 }
 else if s=="." {
 takeIt();take("d");
 while s=="d" takeIt();
 }
 else ERRO();
```

```
if s=="e" {
takeIt();
if s=="+" takeIt()
else if s=="-" takeIt();
take("d");
while s=="d" takeIt();
}
```