

Tópicos em Ciência da Computação

Relatório de Estudo

Aluno: Fernando Antonio Mac Cracken Cezar *cracken@ime.usp.br*

Orientador: Marcelo Finger

1 Objetivo

O objetivo deste plano de estudo foi aprofundar o conhecimento do aluno em alguns tópicos relacionados à proposta de implementação envolvida em sua dissertação de mestrado, ou seja aprofundar os conhecimentos do aluno em tópicos relacionados à implementação de um interpretador SQL.

O primeiro tópico de estudo foi a sintaxe da linguagem SQL, este estudo foi basicamente guiado pelo livro [1]. Posteriormente a gramática da linguagem SQL foi obtida pela URL:

http://ipo53.informatik.htw-dresden.de/XVT/XVTREF45/pwrrref/dbapp_b.htm

e reduzida de acordo com o alvo da pesquisa.

A gramática apresentada nesta URL é a gramática ODBC SQL, que nada mais é que um subconjunto da gramática da linguagem ISO SQL-92. O que não é nenhum problema, visto que o interessante para a pesquisa é apenas um subconjunto da gramática ODBC SQL. Em outras palavras, a pesquisa se interessa por apenas um subconjunto da gramática ODBC SQL que por sua vez está contida na gramática ISO SQL-92. O subconjunto da gramática ISO SQL-92 interessante para a pesquisa encontra-se no Apêndice A.

Possuindo a gramática da linguagem SQL de interesse para pesquisa, foi possível elaborar uma proposta da sintaxe, e conseqüentemente da gramática, da linguagem SQL/OI. SQL/OI é a linguagem SQL com suporte à manipulação de banco de dados obsoletos. Será esta linguagem que o interpretador desenvolvido deverá suportar. A gramática da linguagem SQL/OI encontra-se no Apêndice B.

O próximo tópico de estudo foram as ferramentas lex e yacc, que são respectivamente geradores de analisadores léxicos e sintáticos. Tais ferramentas já são amplamente difundidas e estabelecidas como ferramentas de auxílio para desenvolvimento de compiladores e interpretadores.

Conforme o estudo destas ferramentas foi seguindo, sentiu-se a necessidade de estudar-se também tópicos como implementação de compiladores e interpretadores e linguagens formais, para melhor entendimento das mesmas.

Como último tópico de estudo, estudou-se alternativas de geradores de analisadores léxicos e sintáticos em linguagem Java, visto que o interpretador SQL/OI será implementado em Java.

Os tópicos estudados acima descritos serão discutidos nas próximas seções deste relatório.

2 Introdução

O título da dissertação de mestrado do aluno é *Banco de Dados Obsolescentes e uma proposta de implementação*. Trata-se de uma dissertação envolvida com o projeto SIDAM (Sistema de Informações Distribuídas para Agentes Móveis), projeto este desenvolvido no Instituto de Matemática e Estatística da Universidade de São Paulo.

O projeto SIDAM tem como um exemplo de aplicação um sistema para monitoramento e controle do tráfego de grandes cidades. Este sistema deve manipular informações que se alteram com frequências variáveis com o tempo (por exemplo, dados são alterados com maior frequência em horários de pico durante dias de semana do que tarde da noite durante fins de semana).

As alterações do estado do mundo modelado podem ocorrer a uma frequência muito alta, impossibilitando desta forma que essas alterações sejam imediatamente refletidas no banco de dados. Nestas situações a confiabilidade da informação armazenada decai com o tempo. Esta perda de confiabilidade dos dados armazenados ao passar do tempo é denominada *obsolescência de informação*.

O sistema de monitoramento de tráfego conta também com fontes de informações fixas e móveis (helicópteros e carros), as quais podem não estar disponíveis por certos períodos. Desta forma, o sistema opera em um ambiente suscetível à falhas. Todos estes fatores colocam em evidência a necessidade da preocupação do tratamento de informações obsoletas na modelagem deste sistema.

A proposta da dissertação de mestrado do aluno é tornar a manipulação de banco de dados obsoletos o mais transparente possível. O usuário do banco de dados deve manipular sua base de dados obsoleta como se estivesse manipulando uma base de dados relacional.

A manipulação de uma base de dados obsoleta se dará por meio de um interpretador de comandos SQL desenvolvido pelo aluno. Este interpretador manipulará as tabelas de controle criado em um banco de dados relacional normal. Estas tabelas são responsáveis pelo controle das informações relativas aos dados obsoletos, armazenando assim as funções de decaimento da confiabilidade associada a estes dados.

O interpretador utilizará os valores armazenados nestas tabelas para realizar os cálculos necessários para determinar o grau de confiabilidade de uma informação no momento de sua consulta.

3 Modelagem de informações obsoletas

Banco de dados e base de conhecimentos são desenvolvidos para modelar uma certa parte do mundo. Atualizações nestes repositórios de dados pretendem refletir mudanças do estado do mundo modelado [4]. Existem situações onde a alteração do estado do mundo modelado é muito rápida, de tal forma que não se consegue perceber quando elas ocorrem. Nestas situações a confiabilidade da informação armazenada decai com o tempo. Esta perda de confiabilidade dos dados armazenados ao passar do tempo é denominada *obsolescência de informação*.

Freqüentemente deseja-se modelar um mundo onde seu estado altera-se muito rápido, de tal forma que a obsolescência de informação não pode ser mais desprezada. Sistemas rodando sob este tipo de ambiente devem ser capazes de manipular a queda da confiabilidade das informações a ponto de evitar respostas erradas e comportamentos inesperados.

O tratamento de incerteza de informação é um tema muito abordado nos estudos de Inteligência Artificial e conseqüentemente encontra-se um grande número de trabalhos já desenvolvidos na literatura.

Curiosamente, o tratamento de informações cuja incerteza varia com o tempo não é tão facilmente encontrado na literatura, embora problemas que lidam com tal tipo de informações são encontrados nos mais variados ramos como por exemplo finanças, economia, sociologia, demografia entre outros.

Ao se tentar modelar um sistema de informação para monitoramento de tráfego como o projeto SIDAM, se torna evidente a necessidade do tratamento de informações cuja incerteza varia com o tempo. Este sistema deve manipular informações que mudam dinamicamente e são distribuídas sobre toda a área da cidade de São Paulo e seus arredores. Adicionalmente, o grau de mudança de informação é dependente da localização e do tempo. Por exemplo, os dados sofrem muito mais alterações durante horários de pico nos dias de semana do que à noite ou durante finais de semana.

O monitoramento de tráfego conta também com fontes de informações fixas e móveis (bases e helicópteros), os quais podem não estar disponíveis tempo integral. Todos estes fatores colocam a obsolescência de informação em destaque ao modelar um sistema como o projeto SIDAM.

Informações obsoletas possuem as seguintes características:

- A confiabilidade da informação decai com o tempo;
- O grau de decaimento da confiabilidade pode variar;
- A confiabilidade da informação é inteiramente dependente do momento da última atualização da mesma.

Outra importante característica das informações obsoletas é que sua confiabilidade se propaga para todas as informações derivadas dela.

3.1 Funções de Confiança

Será apresentada nesta seção, algumas definições básicas do que são funções de confiança apenas com o intuito de introdução para apresentar a gramática do SQL/OI na próxima seção. A *credibilidade* ou *confiança* depositada em uma determinada informação A é uma função dependente do tempo $\gamma_A(t) \in [0, 1]$, onde $\gamma_A(t)$ assume o valor 1 (um) quando a informação A reflete perfeitamente o estado do mundo real, já quando a informação A não tem nenhuma relação com a realidade $\gamma_A(t)$ assume o valor 0. A função $\gamma_A(t)$ não é uma medida probabilística, tal tipo de relação entre informações obsoletas e distribuições probabilísticas são tratados em [10].

Define-se também uma credibilidade mínima (*credibility threshold*) associada às informações, ou seja qualquer dado que possua uma credibilidade menor que o valor da credibilidade mínima é considerado um dado não válido.

A obsolescência de todos os dados armazenados em uma base de dados é representada por uma família de funções Γ , por exemplo família de funções lineares, família de funções exponenciais dentre outras.

Da preocupação de um armazenamento, computação e manipulação de funções de confiança de maneira eficiente a família de funções Γ deve seguir quatro princípios apresentados em [6]. Tais princípios tornam o tratamento de funções de confiança não triviais.

4 A linguagem SQL/OI

A linguagem *SQL/OI (Structured Query Language/Obsolescent Information)* é uma proposta para uma extensão da linguagem SQL para tratamento de informações cuja confiabilidade varia com o tempo. Este novo conceito da linguagem SQL faz uso de funções de confiança para determinar se uma certa informação armazenada no banco de dados é considerada confiável no instante de sua consulta.

O modo que a confiabilidade de uma informação varia com o tempo pode ser representado a partir de uma função de confiança (*seção 3.1*). Associa-se uma classe de funções de confiança ao banco de dados no momento que este é convertido para suporte à SQL/OI.

Tendo uma classe de funções de confiança associada ao banco de dados é possível associar uma função de confiança (pertencente a esta classe de funções) às suas tabelas. Esta associação pode ser feita quando uma nova tabela é criada, criando-se assim, uma tabela capaz de manipular informações cuja confiabilidade seja temporalmente dependente. Estas tabelas são denominadas *tabelas obsoletas*, pois linhas destas tabelas podem conter informações obsoletas na hora de sua consulta. As informações armazenadas em tabelas *obsoletas* decaem com o tempo de acordo com sua função de confiabilidade.

A linguagem SQL/OI permite também a criação de tabelas normais (daqui para frente referidas como *tabelas não-obsolentes*), ou seja, tabelas cuja confiabilidade de suas informações não decaiam com o tempo. As informações armazenadas em tabelas não-obsolentes possuem confiabilidade igual a 1 enquanto existirem no banco de dados.

Quando uma tabela obsolente é criada uma função de confiança é associada a ela. Na realidade, associa-se a ela uma classe de funções de confiança, herdada da definição do banco de dados, e um conjunto de parâmetros. Desta forma, definindo uma única função pertencente a esta classe de funções.

Os dados armazenados em uma tabela obsolente compartilham a mesma função de confiança da tabela que os armazena, classe da função e parâmetros que a define. No entanto, é possível alterar estes parâmetros para uma ou mais linhas desta tabela utilizando uma sintaxe do SQL/OI para o comando UPDATE, permitindo que uma nova função de confiança seja associada a estas linhas.

Ao inserir uma nova linha em uma tabela obsolente é possível também associar uma nova função de confiança a esta linha. Pode-se especificar novos parâmetros de definição da função de confiança além dos valores das colunas. Esta especificação é feita utilizando uma sintaxe do SQL/OI para o comando INSERT.

Com o SQL/OI é possível ter uma função de confiança diferente para cada tabela, assim como uma função de confiança diferente para cada linha. A única restrição é que todas as funções de confiança associadas a tabelas de um determinado banco de dados com suporte a SQL/OI, devem pertencer à mesma classe de funções de confiança associada a este banco.

Na próxima seção será discutido a sintaxe do SQL/OI. Como realizar as operações elementares para manipulação de banco de dados e como tratar incerteza de informações com SQL/OI.

4.1.1 Sintaxe dos comandos do SQL/OI

- **Convertendo um banco de dados para ser compatível com SQL/OI**

Para que um determinado banco de dados possua suporte a comandos do SQL/OI é necessário primeiramente convertê-lo a um banco de dados capaz de manipular informações cuja confiabilidade decaia com o tempo.

Esta conversão, na realidade, é apenas uma associação do banco de dados a uma classe de funções de confiança. Esta associação é feita executando-se o seguinte comando:

```
CONVERT DB SET FUNCTION AS <classe de funções de confiança>
```

Executando este comando a classe de funções de confiança <classe de funções de confiança> é associada ao banco de dados corrente.

Por exemplo, para associar a classe de funções exponenciais a um banco de dados executa-se o comando:

```
CONVERT DB SET FUNCTION AS exponencial
```

- **Criando tabelas**

O comando para criação de tabelas do SQL/OI é uma extensão do comando para criação de tabelas do SQL (Apêndice A). O comando de criação de tabelas do SQL/OI permite associar a tabela criada um conjunto de parâmetros que define uma função de confiança. Esta função de confiança pertence à classe de funções de confiança associada ao banco de dados.

A linguagem SQL/OI possui um novo parâmetro na sintaxe do comando **CREATE TABLE** que permite associar uma função de confiança a nova tabela, criando assim uma tabela obsolescente. Este parâmetro é o **CONFIDENCE DECAY PARAMETER**, e necessita como argumento os parâmetros necessários para a definição de uma função de confiança.

Por exemplo, um dado banco de dados A foi associado à classe de funções exponenciais (*tópico anterior*), sabe-se que dois parâmetros (t_0 e β) são necessários e suficientes para definir uma função pertencente à classe de funções exponenciais [6]. Dada uma função de confiança $\alpha(t)$, pertencente à classe de funções exponencial, definida pelos parâmetros $t_0 = 0$ e $\beta = 2$, deseja-se criar uma nova tabela no banco de dados A associada à função $\alpha(t)$ com uma tabela fictícia *estudante*. O comando SQL/OI para criação desta tabela seria:

```
CREATE TABLE ESTUDANTE (  
    NUMERO_EST          CHAR(9),  
    NOME_EST            VARCHAR(50),  
    IDADE               INTEGER,  
    DATA_DE_NASCIMENTO DATE,  
    NUMERO_ESC          CHAR(9),  
  
    PRIMARY KEY (NUMERO_EST),  
    FOREIGN KEY REFERENCES ESCOLA (NUMERO_ESC),  
    CONFIDENCE DECAY PARAMETER (2,0)  
);
```

Para maiores detalhes da sintaxe do comando **CREATE TABLE** do SQL/OI consultar o Apêndice B.

- **Removendo tabelas**

O comando para remoção de tabelas no SQL/OI é igual ao comando de remoção de tabelas do SQL. Para remover uma tabela não é relevante o tratamento de informações cuja confiabilidade decai com o tempo. Para maiores

informações da sintaxe do comando de remoção de tabelas do SQL, e portanto também do SQL/OI.

- **Inserindo linhas em uma tabela**

O comando para inserção de linhas em uma tabela (comando **INSERT**) em sua versão na linguagem SQL/OI além de permitir a especificação dos valores dos campos que serão posteriormente inseridos na tabela permite especificar uma função de confiança associada a estes valores, caso os novos valores sejam inseridos em uma tabela obsolescente.

Uma linha de uma tabela pertencente a um banco de dados capaz de manipular informações obsolescentes pode possuir uma função de confiança associada – tabelas obsolescentes. Esta função deve pertencer à mesma classe de funções de confiança associada à tabela em que esta linha esta contida, conseqüentemente também pertencendo à classe de funções de confiança associada ao banco de dados que contem esta tabela.

Especifica-se esta função por meio de um conjunto de parâmetros que define uma única função da classe de funções de confiança associada à tabela onde se deseja inserir os novos valores. Este conjunto de parâmetros é descrito na mesma linha de comando onde os valores a serem inseridos na tabela são descritos. Esta especificação é realizada pelo parâmetro **CONFIDENCE DECAY PARAMETER** do comando **INSERT** no SQL/OI.

Por exemplo, deseja-se inserir um novo estudante na tabela *estudante*. A ficha do novo estudante é:

Número do Estudante	678
Nome	Eduardo Melo Figueiredo
Data de Nascimento	12/12/1985
Idade	15
Código da Escola	13

Deseja-se associar a este estudante uma função de confiança exponencial $\phi(t)$ (não entremos em detalhes porque uma função de confiança seria associada a um estudante, afinal não conhecemos os estudantes e muito menos as suas habilidades), definida pelos parâmetros $t_0 = 0$ e $\beta = 0.001$.

O comando SQL/OI para inserção deste novo estudante no banco de dados seria:

```
INSERT INTO ESTUDANTE
(NUMERO_EST, NOME_EST, IDADE, DATA_DE_NASCIMENTO, NUMERO_ESC)
VALUES
(678, 'Eduardo Melo Figueiredo', 15, '12-12-1985', 13)
CONFIDENCE DECAY PARAMETER
('beta', 'tzero')
```

```
VALUES  
(0.001, 0);
```

Para maiores detalhes da sintaxe do comando **INSERT** do SQL/OI consultar o Apêndice B.

- **Atualizando linhas de uma tabela**

O comando para atualização de linhas em uma tabela (comando **UPDATE**) em sua versão na linguagem SQL/OI além de ser possível especificar os valores dos campos que serão posteriormente atualizados na tabela é possível alterar a função de confiança associada a estes valores.

O comando **UPDATE** além de associar uma função de confiança a linhas de uma tabela obsolescente, permite também alterar uma função de confiança previamente associada a uma linha de uma tabela obsolescente. Esta alteração é realizada pelo parâmetro **SET CONFIDENCE**.

Por exemplo, deseja-se atualizar a ficha do estudante 678 com os seguintes dados:

Idade	17
Código da Escola	07

Deseja-se alterar também a função de confiança exponencial $\phi(t)$, para uma nova função também exponencial $\phi'(t)$ definida pelos parâmetros $t_0 = 3$ e $\beta = 0.01$.

O comando SQL/OI para alterar a ficha do estudante de número 678 seria:

```
UPDATE ESTUDANTE  
SET IDADE = 17, NUMERO_ESC = 07  
SET CONFIDENCE DECAY beta = 0.01, tzero = 3  
WHERE NUMERO_EST = 678
```

Para maiores detalhes da sintaxe do comando **UPDATE** do SQL/OI consultar o Apêndice B.

- **Removendo linhas de uma tabela**

A novidade no comando **DELETE** do SQL/OI está em um novo predicado na cláusula *search-condition*. Com o novo predicado *confidence predicate* (Apêndice B) é possível remover as linhas de uma tabela obsolescente que obedeçam a uma certa condição sobre o valor da função de confiança naquele momento.

Por exemplo, deseja-se apagar todos os estudantes da tabela estudante cujo valor da função de confiança associada ao estudante no instante t , onde t é o instante da execução do comando **DELETE**, seja menor que 0.003.

```
DELETE FROM ESTUDANTE WHERE CONFIDENCE < 0.003
```

Para maiores detalhes da sintaxe do comando **DELETE** do SQL/OI consultar o Apêndice B.

- **Selecionando linhas de um banco de dados**

A cláusula *search-condition* do comando **SELECT** do SQL/OI possui como novidade também o predicado *confidence predicate*. Com este novo predicado o comando de seleção de linhas é capaz de selecionar linhas de uma tabela obsolescente que obedeçam a uma certa condição sobre o valor da função de confiança naquele momento.

Por exemplo, deseja-se exibir o nome de todos estudantes da tabela estudante cujo valor da função de confiança associada ao estudante no instante t , onde t é o instante da execução do comando **SELECT**, seja maior que 0.5.

```
SELECT NOME_EST FROM ESTUDANTE WHERE CONFIDENCE > 0.5
```

Para maiores detalhes da sintaxe do comando **SELECT** do SQL/OI consultar o Apêndice B.

- **Ajustando parâmetros de sessão e banco de dados**

A linguagem SQL/OI possui um comando que permite ajustar e remover parâmetros de configuração de banco de dados e parâmetros de configuração de sessão.

Os parâmetros de configuração de sessão são removidos ao fechar o cliente SQL/OI, já os parâmetros de banco de dados conservam os seus valores mesmo após o fechamento do cliente SQL/OI.

Os parâmetros de configuração de sessão possuem prioridade sobre os parâmetros de configuração de banco de dados. Caso exista o mesmo parâmetro de configuração ajustado tanto com parâmetro de banco de dados e como parâmetro de sessão em um cliente o valor que será utilizado será o valor do parâmetro de sessão daquele cliente.

Os parâmetros disponíveis para configuração são completamente dependentes da implementação do cliente SQL/OI. Por exemplo, uma implementação pode possuir como um dos parâmetros de configurações disponíveis o parâmetro *threshold* enquanto outra implementação pode possuir este parâmetro como um valor fixo.

Por exemplo, para ajustar um possível parâmetro *threshold* de configuração de banco de dados e sessão respectivamente para o valor 0.1, o comando em SQL/OI seria:

```
SET DB PARAMETER threshold AS 0.1
```

```
SET PARAMETER threshold AS 0.1
```

Para remover o parâmetro de sessão *threshold*, deixando com que o valor do parâmetro *threshold* de banco de dados seja utilizado, o comando em SQL/OI seria:

```
UNSET PARAMETER threshold
```

Para maiores detalhes da sintaxe do comando **SET** e **UNSET** do SQL/OI consultar o Apêndice B.

5 Compiladores e Interpretadores

Um compilador é um tradutor de um programa escrito em uma linguagem, a *linguagem fonte*, para um programa equivalente em uma segunda linguagem, a *linguagem alvo ou objeto*. Tipicamente a linguagem fonte são uma linguagem de programação de alto nível como FORTRAN, Pascal, Ada ou mesmo uma linguagem de nível intermediário como por exemplo a linguagem C, enquanto a linguagem alvo será o código de máquina para a qual o programa foi gerado.

Existem duas maneiras diferentes de executar um programa escrito em uma linguagem de programação de alto nível em um computador. A primeira é traduzir o programa em código de máquina e executá-lo. Este é o processo conhecido como compilação.

A segunda abordagem é escrever um programa capaz de interpretar as linhas de comando de uma linguagem de programação assim que encontradas e executar suas ação(ões) correspondente(s). Tal programa é conhecido como *interpretador*.

Compilação possui a vantagem que a análise e a tradução do código do programa em linguagem de alto nível são realizadas apenas uma vez, embora este processo possa ser, às vezes, demorado. Uma desvantagem do processo de compilação que pode ser apontada é quando o programa desenvolvido apresenta problemas. Neste caso o código de máquina gerado apresentará conseqüentemente também problemas e para encontrá-lo é necessário voltar ao código em linguagem de alto nível até que seja possível cercar e detectar o ponto problemático do programa.

Interpretação possui uma execução do programa muito mais lenta que a execução de um programa compilado. Este fato é facilmente explicável desde que a análise de cada comando da linguagem de alto nível deve ser feita cada vez que o comando é encontrado. No entanto, o processo de detecção de erro de programa se torna mais simples devido ainda estarmos tratando com a linguagem de alto nível durante a execução do programa.

Estas duas abordagens são os extremos, a maioria dos tradutores para a linguagem de máquina é uma mistura das duas abordagens [8].

Para traduzir um programa escrito em uma linguagem de programação para outra, o compilador precisa desmembrar e entender a estrutura da linguagem fonte para posteriormente juntar suas partes de uma maneira diferente

formando o mesmo programa escrito na linguagem destino. É possível dividir um compilador em *front end*, responsável pela análise da estrutura e significado do código fonte; e o *back end*, responsável por gerar o código escrito na linguagem alvo.

Cada uma destas partes ainda pode ser sub dividida em partes menores. O *front end* pode ser dividido em *analizador léxico*, *analizador sintático* e *analizador semântico*. O analisador léxico, algumas vezes denominado *reconhecedor (scanner)*, realiza a análise mais simples. Ele agrupa os símbolos individuais (*caracteres*) do programa fonte em suas respectivas entidades lógicas. Portanto, a sequência de caracteres 'W', 'H', 'I', 'L' e 'E' seria reconhecido como a palavra 'WHILE' e a sequência de caracteres '1', '.' e '0' seriam reconhecidos como um número de ponto flutuante cujo valor é 1.0.

O analisador sintático, comumente denominado *parser*, analisa a estrutura do programa como um todo, agrupando as entidades lógicas reconhecidas pelo *scanner* em construções maiores, como comandos simples, laços de repetição e rotinas, formando a estrutura de todo o programa.

Uma vez que a estrutura do programa foi determinada pode-se analisar então o seu significado (*ou semântica*). Pode-se determinar por exemplo qual variável deve armazenar valores inteiros, e quais valores de ponto flutuante, pode-se também checar se os tamanhos de todos os vetores estão corretamente definidos. Este é o papel do analisador semântico.

É exatamente nesta etapa que o programa é traduzido em uma representação intermediária. O *back-end* do compilador recebe esta representação ele é capaz de gerar o programa na linguagem alvo. Geralmente este processo necessita mais de uma fase.

Primeiramente, um *otimizador de código intermediário* pode transformar a representação intermediária em uma representação intermediária equivalente mais eficiente. Em seguida entra em cena o *gerador de código*, gerando um programa equivalente escrito na linguagem destino. Finalmente, poderá existir um *otimizador de código alvo* para gerar um código alvo final mais eficiente.

O front-end do processo de compilação, ou seja, a análise léxica e a análise sintática serão abordadas com maiores detalhes nas seções 7 e 8, visto que serão estas as etapas utilizadas para a construção do interpretador SQL/OI.

6 Gramáticas Formais

A primeira operação realizada por um compilador é a análise da estrutura do programa fonte. A análise léxica é o nível mais simples de análise, agrupando caracteres simples em entidades básicas. A análise sintática agrupa estas entidades básicas em estruturas que representam o programa fonte completo. *Gramáticas Formais* são usadas para definir a sintaxe de programas escritos na linguagem fonte para os analisadores léxicos e sintáticos.

As análises léxica e sintática são divisões na operação de um compilador essencialmente por motivos de eficiência, ambas utilizam gramáticas formais como base de suas operações.

6.1 Definindo a estrutura de uma linguagem

A sintaxe de uma linguagem é especificada segundo uma abordagem *top-down*. Define-se como cada componente da linguagem é construído partindo do mais simples em direção para o mais complexo. Para tanto, utilizam-se *produções* ou como normalmente denominadas *regras gramaticais*.

A forma geral de uma produção usada para a definição de uma linguagem de programação é:

$$A \rightarrow B_1 B_2 B_3 \dots B_n$$

Esta produção define uma entidade A sendo composta de strings ou entidades mais simples $B_1, B_2, B_3, \dots, B_n$. Esta produção diz que qualquer ocorrência de A no programa pode ser substituída por $B_1, B_2, B_3, \dots, B_n$. Eventualmente, ocorrerá uma string onde mais nada poderá ser substituído, esta string é denominada *sentença*. No contexto de linguagem de programação, programas sintaticamente corretos são sentenças derivadas por uma gramática formal que define a sintaxe da linguagem de programação.

A sintaxe de uma linguagem é definida por uma coleção de produções. A primeira produção deve possuir uma entidade simples da qual todos os programas sintaticamente corretos são derivados:

$$S \rightarrow A_1 A_2 A_3 \dots A_n$$

S é conhecido como *símbolo inicial*, onde todas as sentenças devem ser derivadas de S por sucessivas substituições utilizando as produções da gramática.

É possível ter mais que uma produção dando definições diferentes para um mesmo símbolo:

$$A \rightarrow B_1 B_2 B_3 \dots B_n$$

$$A \rightarrow C_1 C_2 C_3 \dots C_n$$

Tais alternativas podem ser escritas da seguinte maneira:

$$A \rightarrow B_1 B_2 B_3 \dots B_n \mid C_1 C_2 C_3 \dots C_n$$

Produções podem também possuir referências para si próprias, serem recursivas:

$$A \rightarrow Ax \mid y$$

Normalmente é necessário especificar um símbolo que não é substituído por nada. Para indicar esta possibilidade utiliza-se o símbolo *nulo*, ϵ :

$$A \rightarrow B \mid \varepsilon$$

6.2 Definição de Gramática Formal

As produções de uma gramática fazem uso de símbolos. Estes são tantos os símbolos que aparecem nas sentenças de uma gramática, quanto os símbolos que aparecem do lado esquerdo de uma produção definindo grupo de símbolos. Este é o *alfabeto* de uma gramática. Por exemplo, o alfabeto da gramática definida acima é $\{S, A, B, x, y, z\}$.

Um dos símbolos do alfabeto é definido como símbolo inicial, neste caso S . Todas as sentenças devem ser derivadas do símbolo inicial por sucessivas substituições usando as produções da gramática. Tais substituições são realizadas da seguinte forma: um símbolo do lado esquerdo da produção é substituído por uma seqüência de símbolos (opcionalmente seqüência vazia) do lado direito da produção.

O alfabeto de uma gramática pode ser dividido em dois conjuntos distintos. Os *alfabeto terminal* composto por símbolos terminais que são os símbolos que aparecem nas sentenças da linguagem. Os símbolos restantes, os símbolos não terminais formam o *alfabeto não terminal*. No exemplo acima, o alfabeto terminal é composto pelos símbolos $\{x, y, z\}$ e o alfabeto não terminal é composto pelos símbolos $\{S, A, B\}$.

Notando-se V como o conjunto dos símbolos que formam o alfabeto de uma linguagem, com T o alfabeto terminal, por N o alfabeto não terminal e S o símbolo inicial, pode-se definir o conjunto dos alfabetos como:

$$V = T \cup N$$

$$T \cap N = \emptyset$$

Para definir uma linguagem é necessário um conjunto de produções, as quais são definidas da seguinte forma:

$$u \rightarrow v$$

Esta produção pode ser traduzida como: substitua u por v onde quer que u ocorra, onde u e v são seqüência de símbolos de V com u diferente de nulo. Nota-se este conjunto de produções por P .

Pode-se definir formalmente uma gramática como:

Uma gramática G é uma 4-tupla $\{S, P, N, T\}$, onde S é um símbolo inicial, $S \in N$, P um conjunto de produções, N é um conjunto de símbolos não terminais, e T é o conjunto de símbolos terminais.

Define-se uma sentença como:

Uma sentença é uma seqüência de símbolos de T derivadas de S aplicando-se uma ou mais produções de P .

Utilizando a definição de sentença, define-se linguagem como:

A linguagem $L(G)$ é uma linguagem definida por uma gramática G é um conjunto de sentenças derivadas usando G .

6.3 Tipos de Gramáticas

É possível classificar gramáticas em diferentes tipos de acordo com as formas das produções permitidas. Esta classificação é útil, porque é possível obter gramáticas mais simples de serem reconhecidas impondo restrições às produções. A classificação mais difundida é a classificação de Chomsky. Ele sugere quatro tipos de gramática.

Gramáticas de Tipo 0: São as gramáticas mais gerais e também conhecidas como *gramáticas livres*. Produções são da forma $u \rightarrow v$, onde u e v são seqüências arbitrárias de símbolos de V , com u diferente de nulo.

Gramáticas de Tipo 1: Também conhecidas como *gramáticas sensíveis ao contexto*. Produções são da forma $uXw \rightarrow uvw$, onde u, v e w são seqüências arbitrárias de símbolos de V , com v diferente de nulo e X é um único símbolo não terminal. Em outras palavras X pode ser substituído por v , mas somente se for encontrado entre u e w .

Gramáticas de Tipo 2: Também conhecidas como *gramáticas livres de contexto*. Produções são da forma $X \rightarrow v$, onde v é uma seqüência arbitrária de símbolos em V e X é um único não terminal. Em outras palavras X pode ser substituído por v em qualquer situação. Gramáticas usadas pelos analisadores sintáticos em compiladores são do tipo 2.

Gramáticas de Tipo 3: Também conhecidas como *gramáticas finitas ou regulares*. Produções são da forma $X \rightarrow v$ ou $X \rightarrow aY$, onde X e Y são símbolos não terminais e a é um símbolo terminal. Embora estas gramáticas não sejam gerais o suficiente para descreverem uma sintaxe completa de uma linguagem de programação, elas são amplamente utilizadas pelos analisadores léxicos para descrever as entidades básicas que compõe a linguagem.

6.4 Propriedade das Gramáticas

As gramáticas formais possuem propriedades que devem ser conhecidas, entendidas e consideradas ao definir uma gramática para uma linguagem de programação.

6.4.1 Gramáticas Equivalentes

Duas gramáticas G e G' são equivalentes se as linguagens geradas por estas são as mesmas, $L(G) = L(G')$. No entanto, duas gramáticas equivalentes não

precisam necessariamente possuir a mesma árvore de derivação [5] para cada sentença.

6.4.2 Gramáticas Ambíguas

Se uma gramática permite mais que uma mesma sentença seja derivada aplicando-se produções diferentes, esta gramática é chamada de *gramática ambígua*.

Se a árvore de derivação é utilizada posteriormente para fornecer informações a um analisador semântico, gramáticas ambíguas forneceriam sentenças com dois diferentes sentidos, de acordo com as diferentes árvores de derivação para a mesma sentença. Tais gramáticas são um problema para os programadores de compiladores que desejam realizar uma tradução consistente para uma linguagem alvo.

Algumas gramáticas ambíguas podem se tornar não ambíguas introduzindo nesta algumas novas regras [5, W].

6.4.3 Derivações à esquerda e derivações à direita

A análise sintática de uma sentença é realizada por sucessivas aplicações de produções onde o lado esquerdo da produção é substituído pelo seu lado direito. Pode-se existir uma escolha na ordem de como esta substituição é realizada. Por exemplo, a análise sintática da sentença yz utilizando a gramática:

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow Ax \mid y \\ B &\rightarrow z \end{aligned}$$

Utiliza três produções:

$S \rightarrow AB$	resultando na forma AB
$A \rightarrow y$	resultando na forma yB
$B \rightarrow z$	resultando na sentença yz

Nesta derivação primeiramente a produção A foi aplicada seguida da produção B, no entanto outra possibilidade seria aplicar primeiro a produção B, seguida da produção A. É conveniente especificar uma ordem única para construção da árvore de derivação, principalmente se a análise semântica é realizada juntamente com a análise sintática.

6.4.4 Fatoração à esquerda

Freqüentemente encontram-se produções alternativas com primeiros símbolos iguais seguido de um ou mais símbolos distintos entre cada alternativa, por exemplo:

$$X \rightarrow uw \mid uz$$

Pode-se reescrever esta produção por duas novas produções onde não existem produções alternativas com primeiro símbolo igual:

$$A \rightarrow uB$$

$$B \rightarrow w \mid z$$

7 Análise Léxica

O objetivo da análise léxica é transformar uma seqüência de símbolos (*caracteres*) em uma seqüência de *tokens*. Um token léxico é uma seqüência de caracteres que pode ser tratado como uma unidade na gramática que o possui.

A gramática de uma linguagem de programação possui um conjunto finito de tokens, como por exemplo: ID cujos valores podem ser foo, n4, point.

Tomemos como exemplo o fragmento de código fonte em linguagem C abaixo:

```
void foo (int s)
{
    if (s == 1) {
        exit(0);
    }
}
```

Um analisador léxico ao analisar este código poderia retornar:

```
VOID ID(foo) LPAREN INTEGER ID(s) RPAREN LBRACE
IF LPAREN ID(s) EQUAL NUM(1) RPAREN LBRACE EXIT LPAREN
NUM(0) RPAREN SEMICOMA RBRACE RBRACE
```

onde o tipo de cada token é indicado; identificadores e literais possuem um valor associado.

A ferramenta mais utilizada na especificação de tokens é *expressões regulares*.

7.1 Expressões Regulares

Seja uma linguagem um conjunto de palavras, uma palavra um conjunto finito de símbolos, onde estes símbolos pertencem a um alfabeto. Ainda mais, esta linguagem não possui significado associado às palavras.

Deseja-se apenas classificar palavras como pertencentes ou não a esta linguagem. A melhor ferramenta para especificar tal tipo de linguagem é

expressões regulares. Cada expressão especifica um conjunto (finito ou não) de palavras.

Símbolo: para cada símbolo **a** do alfabeto da linguagem, a expressão regular **a** especifica a linguagem contendo somente a palavra **a**.

Alternação: Dada duas expressões regulares **M** e **N**, a operação de alternância denotada com uma barra vertical (|) produz uma nova expressão regular **M | N**. Uma palavra está na linguagem definida por **M | N** se ela está na linguagem definida por **M** ou na linguagem definida por **N**. Portanto, a linguagem **a | b** contém as duas palavras **a** e **b**.

Concatenação: Dadas duas expressões regulares **M** e **N**, o operador de concatenação \cdot produz uma nova expressão regular **M · N**. Uma palavra pertence a linguagem definida por **M · N** se ela é a concatenação de duas palavras **a** e **b**, tal que **a** é uma palavra pertencente a linguagem definida por **M** e **b** é uma palavra pertencente a linguagem definida por **N**. Desta forma, a expressão regular **(a | b) · a** define uma linguagem que contém duas palavras **aa** e **ba**.

Epsilon: A expressão regular ϵ representa uma linguagem cuja única palavra é a palavra vazia. Portanto, **(a · b) | ϵ** representa a linguagem {“, “ab”}.

Repetição: Dada uma expressão regular **M**, seu fecho de Kleene é **M***. Uma palavra está em **M***, se esta é uma concatenação de zero ou mais palavras de **M**. Portanto, **((a | b) · a)*** representa o conjunto infinito {“, “aa”, “ba”, “aaaa”, “baaa”, “aaba”, “baba”, “aaaaa”, ...}.

Usando símbolo, alternância, concatenação, epsilon e repetição pode-se especificar o conjunto dos caracteres ASCII correspondente aos tokens da linguagem de programação. Alguns exemplos simples:

$(0 | 1)^* \cdot 0$ Números binários múltiplos de dois.

b*(abb*)*(a| ϵ) Palavras de a's e b's com a's não consecutivos

Ao escrever expressões regulares, é comum omitir o símbolo de concatenação e o epsilon, e assume-se que repetições têm prioridade sobre concatenações e concatenações têm prioridade sobre alternâncias.

Algumas abreviações são amplamente utilizadas: **[abcd]** significa **(a|b|c|d)**, **[b-g]** significa **[bcdefg]**, **[b-gM-Qkr]** significa **[bcdefgMNO PQkr]**, **M?** significa **(M| ϵ)** e **M+** significa **(M · M*)**.

Expressões regulares são exatamente equivalentes à gramáticas do tipo 3. Qualquer linguagem do tipo 3 pode ser descrita por uma expressão regular e vice-versa. Maiores informações sobre expressões regulares em [9].

7.2 Autômatos finitos

Expressões regulares são convenientes para especificar tokens léxicos, mas se faz necessário um formalismo capaz de ser implementado em computador. Para tal, utiliza-se os autômatos finitos. Um autômato finito possui um conjunto finito de *estados*; também possui *arestas* que conectam um estado ao outro

onde cada uma possui um caractere associado. Um autômato finito possui um estado especial o chamado *estado inicial* e pode possuir um ou mais *estados finais*. Um autômato finito tem como entrada uma seqüência de caracteres, e produz como saída informação sobre a seqüência de caracteres de entrada.

O autômato finito começa a sua execução no estado inicial, conforme caracteres são introduzidos em sua entrada o autômato se move de um estado para estado seguindo a aresta associada com o caractere recém recebido. Se no final da seqüência de caracteres o autômato se encontra em um de seus estados finais a seqüência de entrada é dita como uma seqüência válida e é considerada uma seqüência não válida caso contrário.

Um autômato finito determinístico (DFA) é um autômato finito que não possui duas arestas associadas ao mesmo caractere saindo de um mesmo estado, enquanto nos autômatos finitos não determinísticos (NFA) esta regra não se aplica.

Autômatos finitos determinísticos são facilmente representados em uma linguagem de programação por um vetor bidimensional indexado por número de estados do autômato e caracteres possíveis de entrada [3, 5, 8].

Existem algoritmos simples e muito eficientes capazes de transformar uma expressão regular em um autômato finito não determinístico, que por sua vez pode ser transformado, também facilmente por meio de um algoritmo, em um autômato finito determinístico [3, 5]. Desta forma uma expressão regular é facilmente mapeada a um autômato finito determinístico que por sua vez é facilmente mapeado a um vetor bi-dimensional. Assim expressões regulares podem especificar um programa capaz de realizar análise léxica de uma linguagem, tal programa recebe como entrada um código fonte e emite como saída tokens desta linguagem.

8 Análise Sintática

Durante a análise sintática determina-se se uma determinada seqüência de tokens fornecida pelo analisador léxico é uma sentença válida da linguagem ou não.

Um dos critérios básicos para um algoritmo de análise sintática é que este seja eficiente. Em geral, deseja-se que o algoritmo possua tempo de análise proporcional ao tamanho do arquivo analisado. Um outro critério relacionado à eficiência é que o analisador seja capaz de determinar suas ações considerando somente um número fixo de tokens k à frente do ponto corrente de análise. Algoritmos de análise sintática efetivos possuem $k=1$. Os algoritmos de análise sintática não devem conter *back-tracking*, estes devem operar deterministicamente durante toda a sua execução.

Existem vários métodos de análise sintática, mas é claro que o método de análise deve ser poderoso o suficiente para analisar a linguagem desejada [5]. Diferentes métodos de análise são apropriados para diferentes subconjuntos de gramática de tipo 2. Se um analisador sintático não é adequado para uma

determinada gramática ou um diferente analisador sintático deve ser utilizado ou a gramática deve ser reescrita.

Algumas gramáticas são fáceis de serem sintaticamente analisadas utilizando-se um algoritmo conhecido com *descendente recursivo*. Em linhas gerais este algoritmo transforma cada produção da gramática em uma cláusula condicional de uma função recursiva. Um analisador sintático descendente recursivo possui uma função para cada símbolo não terminal e uma cláusula condicional para cada produção [3].

Analisadores sintáticos descendentes recursivos funcionam somente em gramáticas onde o primeiro símbolo terminal de cada produção prove informação suficiente para a escolha da próxima produção a ser utilizada. Para melhor entendimento será definido o conjunto FIRST, deste conjunto é possível derivar um analisador sintático descendente recursivo livre de conflito utilizando-se um simples algoritmo como apresentado em [3].

8.1 Conjuntos FIRST e FOLLOW

Dada uma palavra γ contendo símbolos terminais e não terminais, $FIRST(\gamma)$ é o conjunto de todos os símbolos terminais que podem começar qualquer palavra derivada de γ .

Se duas produções $X \rightarrow \gamma_1$ e $X \rightarrow \gamma_2$ possuem o mesmo símbolo X à esquerda da produção e seus símbolos à direita possuem conjuntos FIRST sobrepostos, então esta gramática não pode ser analisada utilizando algoritmo descendente recursivo. Se um símbolo terminal I pertence a $FIRST(\gamma_1)$ e à $FIRST(\gamma_2)$, então a função que representa a produção X em um analisador descendente recursivo não saberá qual cláusula condicional executar quando o token de entrada é I .

O cálculo do conjunto FIRST aparentemente é muito simples: se $\gamma = XYZ$, aparentemente Y e Z podem ser ignorados, e $FIRST(X)$ é a única coisa que realmente interessa para o cálculo de $FIRST(\gamma)$. Infelizmente as coisas não são tão simples assim, como por exemplo na gramática apresentada logo abaixo:

$$\begin{array}{ll} Z \rightarrow d & Y \rightarrow \epsilon \\ Z \rightarrow XYZ & X \rightarrow Y \\ Y \rightarrow c & X \rightarrow a \end{array}$$

Nesta gramática Y pode produzir a palavra vazia – e portanto X pode produzir a palavra vazia – descobre-se que $FIRST(XYZ)$ deve incluir $FIRST(Z)$. Portanto, no cálculo dos conjuntos FIRST, é necessário armazenar quais símbolos poderão produzir palavras vazias; tais símbolos são denominados *nullable*. É necessário armazenar também o que pode seguir um símbolo *nullable*.

Dada uma palavra particular γ de uma gramática contendo símbolos terminais e não terminais,

- $nullabe(X)$ é verdadeiro se X pode derivar a palavra vazia

- $FIRST(\gamma)$ é o conjunto de terminais que podem iniciar palavras derivadas de γ
- $FOLLOW(X)$ é o conjunto de terminais que podem imediatamente seguir X . Isto é $t \in FOLLOW(X)$ se existe qualquer derivação contendo Xt . Isto pode ocorrer se a derivação $XYZt$ onde Y e Z ambos derivem ϵ .

Um algoritmo capaz de gerar um tabela com os conjuntos nullabe, $FIRST$ e $FOLLOW$ de uma gramática é apresentado em [3].

8.2 Analisadores Sintáticos Descendente Recursivo

Considere um analisador sintático descendente recursivo. A função de análise para algum não terminal X possui uma cláusula condicional para cada produção de X ; o analisador precisa decidir qual destas cláusulas condicionais escolher dependendo exclusivamente do próximo token T de entrada. Se a escolha da produção é correta para cada (X, T) , então a escrita de um analisador sintático descendente recursivo é possível. Toda a informação necessária pode ser mapeada em uma tabela bidimensional indexada por símbolos não terminais x terminais [3].

A tabela abaixo representa a gramática logo acima:

	A	C	D
X	$X \rightarrow a$	$X \rightarrow Y$	$X \rightarrow Y$
Y	$Y \rightarrow$	$Y \rightarrow \epsilon$ $Y \rightarrow c$	$Y \rightarrow \epsilon$
Z	$Z \rightarrow XYZ$	$Z \rightarrow XYZ$	$Z \rightarrow d$ $Z \rightarrow XYZ$

Nota-se que algumas das entradas da tabela possuem mais de uma produção, isto significa que esta gramática não pode ser analisada utilizando-se um analisador sintático descendente recursivo. Esta gramática é uma gramática ambígua, gramática ambíguas produzem mais de uma entrada em tabelas descendente recursivas.

Gramáticas cuja tabela descendente recursiva não possuem entradas duplicadas são chamadas gramáticas $LL(1)$. Isto significa que os analisadores sintáticos descendentes recursivos analisam a entrada da esquerda para a direita, realizando derivações à esquerda olhando apenas um símbolo à frente (*left-to-right parse, leftmost-derivation, 1-symbol lookahead*).

8.3 Análise Sintática LR

Para realizar uma análise sintática $LL(k)$ é necessário saber qual produção utilizar apenas olhando os primeiros k tokens do lado direito da produção. Uma técnica mais poderosa é a técnica de análise $LR(k)$. Nesta técnica a decisão é

adiada até que todos os tokens do lado direito da produção em questão sejam encontrados na entrada (e mais k tokens à frente).

LR(k) significa que os analisadores sintáticos LR(k) analisam a entrada da esquerda para a direita, realizando derivações à direita olhando k símbolo à frente.

Um analisador LR(k) possui uma pilha e uma entrada. Os primeiros k tokens da entrada são os tokens que devem ser checados à frente. Baseado no conteúdo da pilha e nos tokens que são verificados à frente, o analisador sintático pode realizar duas ações:

Shift: coloque o primeiro token da entrada no topo da pilha

Reduce: escolhida uma produção $X \rightarrow ABC$; retire C, B, A do topo da pilha e coloque X.

Maiores detalhes de como esta pilha é interpretada (por meio de um DFA) podem ser conseguidos em [3, 8].

Gramáticas LR(0) são aquelas que podem ser analisadas utilizando-se somente a pilha, realizando-se decisões de shift/reduce sem a necessidade de olhar tokens à frente. Esta classe de gramática é muito fraca para ser realmente útil.

Uma classe de gramática mais poderosa é a classe LR(1), a maioria das linguagens de programação cujas sintaxes podem ser descritas por uma linguagem livre de contexto possuem uma gramática LR(1). O algoritmo de análise sintática para gramáticas LR(1) assim como seu aperfeiçoamento denominado LALR(1) pode ser encontrado em [3].

9 lex & yacc

Duas das mais úteis abstrações utilizadas nos compiladores modernos são *gramáticas livres de contexto* [seção 3.4], para análise sintática, e expressões regulares [seção 4.1], para análise léxica. Ferramentas foram desenvolvidas para facilitar a manipulação de tais abstrações, como o *Yacc* – capaz de converter uma gramática livre de contexto LR em um analisador sintático e o *Lex* – capaz de converter uma especificação declarativa em um programa de análise léxica.

9.1 Lex – O gerador automático de analisadores léxicos

Lex é um gerador automático de analisadores léxicos extensivamente utilizado no mundo Unix, embora este não seja limitado apenas a este tipo de sistema operacional.

Um arquivo de especificação *lex* contém uma série de expressões regulares as quais serão comparadas com a seqüência de caracteres de entrada. Cada vez que o padrão de uma expressão regular é encontrado, o programa lex executa um código C, previamente especificado, que poderá ou não tratar o texto encontrado. Desta maneira um programa lex divide a seqüência de caracteres de entrada em tokens. Esta seqüência de tokens pode ser posteriormente utilizada pelo *YACC* [seção 6.2] ou pode ser realmente o “produto final”.

O programa lex em si não produz um programa executável; ele traduz o arquivo de especificação lex em um programa em linguagem C que contém uma rotina denominada *yylex()*. Outros programas devem chamar esta rotina para chamar o analisador léxico, a rotina *yylex()* consome um token de cada vez.

9.1.1 Arquivo de especificação Lex

Um arquivo de especificação Lex pode ser dividido em três áreas:

```
declarações
%%
produções
%%
código adicional
```

nenhuma das áreas é obrigatória, no entanto o primeiro %% é. Este marca a divisão entre as áreas de declarações e de produções.

- **Primeira Área: Declarações**

Esta área do arquivo de especificação lex pode conter:

1. Código escrito na linguagem destino (C ou C++), cercado por %{ e %}, o qual será inserido no topo do arquivo fonte C que o lex criará. Este é geralmente onde os includes são inseridos.
2. Expressões regulares, definindo símbolos não terminais como, letras, dígitos. Esta especificação possui a forma:

não terminal *expressão regular*

Estas definições podem ser utilizadas no final desta área do arquivo ou na segunda área cercadas por chaves {}.

Exemplo:

```
%{
#include "calc.h"
#include <stdio.h>

#include <stdlib.h>
}%

/* Regular expressions */
/* ----- */

white      [\t\n ]+
letter     [A-Za-z]
```

```

digit10      [0-9]                /* base 10 */
digit16      [0-9A-Fa-f]         /* base 16 */

identifier   {letter}(_|{letter}|{digit10})*
int10        {digit10}+

```

- **Segunda Área: Produções**

Esta área do arquivo de especificação lex instrui ao lex o que fazer quando um padrão que casa com alguma expressão regular especificada é encontrado:

Esta área do arquivo de especificação lex pode conter:

1. Código escrito na linguagem destino (C ou C++), cercado por %{ e %} (no começo da linha). Estas especificações serão inseridas no começo da função *yylex()*, a qual é a função que consome os tokens e retorna inteiros.
2. Produções possuem a sintaxe:

expressão regular *ação*

Se a *ação* estiver faltando, lex irá imprimir os caracteres encontrados na saída padrão. Se a ação é especificada, esta deve ser escrita no arquivo em linguagem C gerado. Se a ação contém mais que uma instrução ou ela deve ser escrita em mais de uma linha ou cercada por chaves {}.

Finalmente, a variável *yytext* pode ser usada nas dentro das ações possuindo os caracteres aceito pela expressão regular.

Exemplo:

```

%%

[ \t]+$      ;
[ \t]        printf(" ");

```

- **Terceira Área: Código Adicional**

Adiciona-se aqui qualquer código extra que se faça necessário.

Maiores detalhes sobre lex podem ser obtidos em [2]. O programa lex possui uma versão GNU, o Flex. Maiores detalhes podem ser obtidos em:

<http://www.gnu.org/software/flex/flex.html>

9.2 Yacc – O gerador automático de analisadores sintáticos

Yacc (Yet Another Compiler Compiler) recebe uma especificação de uma gramática de uma linguagem de programação e produz um analisador sintático

LALR(1). O programa fonte fornecido ao analisador sintático posteriormente deve ser uma seqüência de tokens, portanto um analisador léxico deve ser desenvolvido separadamente (normalmente utilizando o lex).

9.2.1 Arquivo de especificação Yacc

Um arquivo de especificação yacc, assim como um arquivo de especificação lex, pode ser dividido em três áreas:

```
declarações
%%
produções
%%
código adicional
```

sendo que somente o primeiro %% e a segunda parte são mandatórias.

- **Primeira Área: Declarações**

Esta área do arquivo de especificação yacc pode conter:

1. Código escrito na linguagem destino (C ou C++), cercado por %{ e %} (cada símbolo no começo da linha) , o qual será inserido no topo do arquivo fonte C que o yacc criará. Este é geralmente onde os includes são inseridos.
2. Declarações dos tokens que podem ser encontrados:

%token TOKEN

3. O tipo do terminal, utilizando a palavra reservada %union
4. Informações sobre prioridade de operadores e associatividade
5. O símbolo inicial da gramática com a palavra reservada %start (se não especificado a primeira produção da segunda área do arquivo será utilizada)

A variável yylval, declarada como %union, é muito importante neste arquivo deste é nesta variável que a descrição do último token lido é armazenada.

- **Segunda Área: Produções**

Esta área não pode ser vazia. Ela deve conter:

1. Declarações e/ou definições cercadas por %{ %}.
2. Produções da gramática da linguagem:

```
simbolo_nao_terminal:
                                simbolo_1      { acao_1 }
```



```

| simbolo_2          { acao_2 }
| ...
| simbolo_n          { acao_n }
;

```

- **Terceira Área: Código Adicional**

Esta área contém o código adicional, ela deve conter uma função `main()` (a qual deve chamar a função `yyparse`), e uma função `yyerror` (`char *message`), a qual é chamada quando um erro de sintaxe é encontrado.

Maiores detalhes sobre yacc podem ser obtidos em [2]. O programa yacc possui uma versão GNU, o Bison. Maiores detalhes podem ser obtidos em:

<http://www.gnu.org/software/bison/bison.html>

10 Geradores automáticos de analisadores léxicos e sintáticos em linguagem Java

Esta seção apresentará em linhas gerais alguns dos geradores automáticos de analisadores léxicos e sintáticos em linguagem Java disponíveis. Vários pacotes foram encontrados, mas os que se mostraram mais interessantes por serem, até certo ponto, compatíveis com lex ou com yacc são mostrados a seguir. Uma boa referência para ferramentas de desenvolvimento de compiladores em Java é:

<http://www.compilerconstruction.org/catalog/java.html>

10.1 JLEX - <http://www.cs.princeton.edu/~appel/modern/java/JLex/>

O gerador de analisador léxico JLex é baseado principalmente no modelo do Lex. JLex lê um arquivo JLex e gera o código fonte do analisador léxico correspondente escrito em linguagem Java.

10.1.1 Arquivo de especificação JLex

O arquivo de especificação JLex assim como o arquivo de especificação lex pode ser dividido em três áreas:

```

código do usuário
%%
diretivas JLex
%%
expressões regulares

```

O arquivo de especificação do JLex à primeira vista parece ser muito parecido com o arquivo de especificação do lex, mas na verdade não são. O conteúdo da primeira seção do arquivo JLex também é copiado para o topo do

arquivo fonte gerado, assim como no arquivo de configuração lex. No entanto a primeira seção do arquivo de configuração JLex não possui definições de símbolos não terminais, estas definições são realizadas na segunda área do arquivo.

É na segunda área do arquivo de especificação JLex onde se encontram também as diretivas JLex. As diretivas JLex definem alguns parâmetros que modificam o comportamento do analisador léxico.

E obviamente na terceira área encontram-se as expressões regulares que definem os tokens aceitos pelos analisadores léxico

Maiores detalhes sobre JLex assim como a sintaxe do seu arquivo de especificação pode ser encontrada na URL:

<http://www.cs.princeton.edu/~appel/modern/java/JLex/>

10.2 JFLEX - *<http://www.jflex.de/>*

JFlex é um gerador de analisadores léxicos para Java escrito em Java. Este foi completamente baseado na ferramenta JLex.

Tendo em vista que JFlex é completamente baseado no JLex (seção anterior) possuindo todas as suas características , nesta seção serão apresentadas apenas as vantagens do JFlex sobre o JLex.

- Analisadores léxicos gerados mais rápidos
- Criação do analisador léxico mais rápida
- Três tipos de código podem ser gerados a fim de obter a melhor relação desempenho/tamanho
- Classes de caracteres já pré-definidas
- Comentários são permitidos por todo o arquivo de especificação
- Definições de macro são expressões regulares e não só texto.
- Caractere de fim de linha independente de plataforma (operador "\$")
- Agrupamento de regras com o mesmo estado léxico
- Suporte a depuração
- Contador de colunas

Maiores detalhes sobre JFlex assim como uma documentação de boa qualidade pode ser encontrada na URL:

<http://www.jflex.de/>

10.3 BYACC/Java - *<http://troi.lincom-asg.com/~rjamison/byacc/>*

BYACC/Java é uma extensão do gerador de analisador sintático Berkeley v 1.8 YACC-compatível. O programa BYACC lê um arquivo fonte BYACC, e gera

um ou mais arquivos fontes em linguagem C, os quais após corretamente compilados produzirão um parser de gramática LALR.

O programa BYACC possui uma diretiva de linha de comando `-j` a qual requisitará ao BYACC gerar arquivos fontes em linguagem Java no lugar de arquivos fontes em linguagem C.

O projeto original do YACC data de aproximadamente vinte anos atrás, obviamente novas e melhores tecnologias foram desenvolvidas desde então. Ainda o desenvolvimento de tal tipo de ferramenta para a linguagem Java possui inúmeras vantagens:

- BYACC/Java é codificado em C, portanto a geração de código Java é extremamente rápida.
- O bytecode resultante do código Java gerado é consideravelmente pequeno – começando em 11 kbytes.
- Nenhum runtime system é necessário. O código fonte gerado é um parser completo.
- BYACC é compatível com as gramáticas YACC já existentes, tornando a portabilidade para a linguagem Java de uma base de código fonte baseado em YACC mais fácil.
- Muitos desenvolvedores já estão familiarizados com YACC.
- É absolutamente grátis

Para que o BYACC gere um código fonte em Java de uma especificação YACC em Java basta executar o comando:

```
yacc -J arquivo.y
```

Este comando gerará um arquivo cujo nome é *parser.java* o qual pode ser compilado com o comando *javac parser.java*.

A especificação do arquivo BYACC para geração de um parser em Java segue a mesma especificação para a geração de um parser em Linguagem C, com a diferença que as ações escritas nesta especificação devem ser escritas em Java.

Maiores detalhes sobre BYACC assim como o exemplo clássico da calculadora em BYACC para Java pode ser encontrado na URL:

<http://troi.lincom-asg.com/~rjamison/byacc/>

10.4 CUP - <http://www.cs.princeton.edu/~appel/modern/java/CUP/index.html>

CUP (Constructor of Useful Parsers) é um gerador de analisadores sintático LALR. Esta ferramenta segue a mesma linha do famoso programa yacc (seção 6.2), oferecendo ainda algumas características adicionais. No entanto CUP é escrito em Java, seu arquivo de especificação contém trechos de código em Java e o analisador sintático produzido por ele é escrito em Java.

A utilização do CUP envolve a criação de um arquivo de especificação contendo a gramática para a qual o analisador sintático será utilizado e um analisador léxico capaz de gerar tokens relevantes para alimentar o analisador sintático em questão.

O arquivo de especificação CUP pode ser dividido em quatro áreas:

A primeira área contém declarações preliminares que definem como o analisador sintático deve ser gerador, e especifica a biblioteca de *runtime* necessária. A segunda área define os símbolos terminais e não terminais, e associa classes de objetos a estes. A terceira área define a precedência e associatividade de terminais e a quarta e última área contém a especificação da gramática em si.

Maiores detalhes sobre CUP podem ser encontrados na URL:

<http://www.cs.princeton.edu/~appel/modern/java/CUP/index.html>

11 Conclusão

O estudo foi realmente efetivo para a compreensão de como o interpretador SQL/OI será implementado.

As teorias sobre linguagens formais e gramáticas foram utilizadas ao reduzir a linguagem SQL, assim como na criação da proposta de gramática da linguagem SQL/OI.

O estudo das ferramentas lex e yacc foi de fundamental ajuda para o entendimento do funcionamento dos seus ‘parceiros’ em linguagem Java. Todas as documentações das ferramentas para auxílio de desenvolvimento de compiladores em Java referem-se ao lex ou yacc.

A ferramentas escolhidas para o desenvolvimento do interpretador SQL/OI foram JFlex como gerador de analisador léxico e BYacc como gerador de analisadores léxicos.

A ferramenta JFlex foi o gerador de analisadores léxicos de escolha, pois esta se mostrou a ferramenta mais compatível com o seu primo “primata” lex, além de possuir uma boa documentação incluindo explicações de como portar arquivos lex para JFlex, o que é de extrema ajuda visto que [2] possui o arquivo de especificação lex para a linguagem SQL.

A ferramenta Byacc foi o gerador de analisadores sintáticos de escolha, pois esta ferramenta é a mais compatível com yacc, o que é de extrema ajuda visto que [2] possui também o arquivo de especificação yacc para a linguagem SQL. Além disto Byacc não necessita de um sistema de runtime como Java CUP.

12 Referências

- [1] R. Elmasri / S. B. Navathe. *Fundamentals of Database Systems – Second Edition*. The Benjamin/Cummings Publishing Company, Inc. 1994
- [2] J. R. Levine / T. Mason / D. Brown. *Lex & Yacc – Second Edition*. O'Reilly & Associates, Inc. 1992
- [3] A. W. Appel . *Modern Compiler Implementation in Java*. Cambridge University Press, 1998
- [4] F.S.C.da Silva, M. Finger. *Temporal data obsolescence: Modelling problems*. In Proceeding of the 5th International Workshop on Temporal Representation and Reasoning (TIME'98), Sanibel Island, Florida 1998. IEEE Computer Science Press
- [5] J.P.Bennett. *Introduction To Compiling Techniques*. McGraw-Hill International, 1990
- [6] M.Finger. *Principles for Modelling Obsolescent Information*
- [7] J.E.Hopcroft, Ullman, Rotwani. *Introduction to Automata, Theory, Languages and Computation*. Addison-Wesley – 2nd edition - 2000
- [8] Aho, Alfred V. Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [9] J.R.F. Friedl. *Mastering Regular Expressions*. O'Reilly & Associates, Inc. 1997
- [10] S.A.Vitório, *Obsolescência de Informação*