**Regular Expression**

**R+:** one or more strings from L(R ), R(R\*)

**R?:** optional R: (R|e)

**[abce]** one of the listed characters: (a|b|c|e)

**[a-z]** one character from this range: (a|b|c|d|e|…|y|z)

**[^ab]** anything but one of the listed chars

**[^a-z]** one character not from this range

**1.Numbers**

**nat =** [0-9]+

**signedNat =** (+|-)?nat

**number =** signedNat(“．”nat)? (E signedNat)?

**2.Reserved Words and Identifiers**

**reserved =** if | while | do |………

**letter =** [a-z A-Z]

**digit =** [0-9]

**identifier =** letter (letter | digit)\*

**3. Comment**

Several forms:

/\* this is a C comment \*/ ba((ab))\*ab (wrong)

″/\*″ ([^\*/] | [^\*] ″/″|″\*″[^/]) \* ″\*/″

not include /\*/。。。。。

not include /\*\*\*/

″/\*″″/″\* ([^\*/] | [^\*]″/ ″ | ″\* ″[^/]) \* ″\* ″ \* ″ \*/ ″

{ this is a pascal comment } {( } )\*}

; this is a schema comment

-- this is an Ada comment --(newline)\*

**From an NFA to a DFA**

**1.the e-closure of a Set of states:**

the e-closure of a single state s is the set of states reachable by a series of zero or more -transitions.

the e-closure of a set of states : the union of the -closures of each individual state.

**2. the Subset Construction:**

1) Compute the e-closure of the start state of M; this becomes the start state.

2) Given a set S of states and a character a in the alphabet, compute the set S’a = { t | for some s in S there is a transition from s to t on a }. Then, compute , the e-closure of .

3) Continue with this process until no new states or transitions are created. Mark as accepting those states constructed in this manner that contain an accepting state of M.

**Minimizing the number of states in a DFA**

Given the algorithm as follow:

1.It begins with the most optimistic assumption possible: it creates two sets

One consisting of all the accepting states

The other consisting of all the nonaccepting states.

2.Given this partition of the states of the original DFA, consider the transitions on each character a of the alphabet.

If all accepting states have transitions on a to accepting states.

defines an a-transition from the new accepting state (the set of all the old accepting states) to itself.

If all accepting states have transitions on a to nonaccepting states

defines an a-transition from the new accepting state to the new nonaccepting state (the set of all the old nonaccepting stales).

2 Given this partition of the states of the original DFA, consider the transitions on each character a of the alphabet.

If there are two accepting states s and t that have transitions on a that land in different sets,

no a-transition can be defined for this grouping of the states. We say that a distinguishes the states s and t.

If there are two accepting states s and t such that s has an a-transition to another accepting state, while t has no a-transition at all (i.e., an error transition) ,

then a distinguishes s and t.

I f any further sets are split, we must return and repeat the process from the beginning.

This process continues until

(1) all sets contain only one element (in which case, we have shown the original DFA to be minimal)

(2) until no further splitting of sets occurs.

**Context-free grammar**

**Left Recursive**: the nonterminal A appears as the first symbol on the right-hand side of the rule defining A.

**Right Recursive**: the nonterminal A appears as the last symbol on the right-hand side of the rule defining A.

**Parse Trees**

**A left­most derivation**: a derivation in which the leftmost nonterminal is replaced at each step in the derivation.

Corresponds to the preorder numbering of the internal nodes of its associated parse tree.

**A rightmost derivation**: a derivation in which the rightmost nonterminal is replaced at each step in the derivation.

Corresponds to the postorder numbering of the internal nodes of its associated parse tree.

**Remove ambiguity**

State a disambiguating rule that establishes the relative precedences of the three operations represented.

The associativity of each of the operations of addition, subtraction, and multiplication

**Precedence and associativity 优先权和结合性**

**优先权：**Group the operators into groups of equal precedence, and for each precedence we must write a different rule.

For example, the precedence of multiplication over addition and subtraction can be added to our simple expression grammar as follows:

exp -> exp addop exp | term

addop  -> + | -

term -> term mulop term| factor

mulop  -> \*

factor -> ( exp ) | number

在这个文法中，乘法被归结在term规则下，而加法和减法被归在exp规则下，由于exp的基本情况是term，这就意味着加法和减法在分析树和语法树中将被表现地更高一些（也就是更接近于根）因此也就接受了更低一级的优先权。

A precedence cascade: a grouping of operators into different precedence levels is a standard method in syntactic specification using BNF.

The precedence cascades cause the parse trees to become much more complex. The syntax trees, however, are not affected.

优先级联使得分析树更加复杂，但是语法树不受影响

**The dangling else problem悬挂else问题**

Which one is correct depends on whether we want to associate the single else-part with the first or the second if-statement:

the first parse tree associates the else-part with the first if-statement;

the second parse tree associates it with the second if-statement.

An else-part should always be associated with the nearest if-statement that does not yet have an associated else-part. Called the **most closely nested rule**

**Inessential ambiguity 无关紧要的二义性**

**Inessential ambiguity**: the associated semantics do not depend on what disambiguating rule is used.

Eg: Arithmetic addition or string concatenation.

that represent associative operations

(a binary operator • is associative if (a • b) • c = a • (b • c) for all values a, b, and c).

The syntax trees are still distinct, semantic value are the same. 语法树不同，但语义分析是相同的。

**EBNF和语法图**

**Top-Down Parsing**

**LL(1) Parsing**

The two actions:

(1) Generate: replace a non-terminal A at the top of the stack by a string α(in reverse) using a grammar rule A →α,

(2) Match: match a token on top of the stack with the next input token.

**LL(1) Parsing Table and algorithm**

对每个非终结符-记号给出唯一的选择

如果文法G相关的LL(1)分析表的每个项目中之多只有一个产生式，则该文法就是LL(1)文法，即LL1没有二义性

The general LL(1) Parsing table definition:

The table is a two-dimensional array indexed by non-terminals and terminals

The table contains production choices to use at the appropriate parsing step, which called M[N,T].

N is the set of non-terminals of the grammar;

T is the set of terminals or tokens (including $);

Any entrances remaining empty represent potential errors.

**Left Recursion Removal and Left Factoring**

**Case1: Simple immediate left recursion 简单直接左递归**

A -> A α| β

Rewrite this grammar rule into two rules;

A –> βA’ A’ -> αA’ | e

Example: exp → exp addop term | term

exp → term exp’

exp’ → addop term exp’ | ε

**Case 2: General immediate left recursion**

A → A 1 | A 2 | … | A n |β1|β2|…|βm

Where none of β1,…, βm begin with A.

The solution is similar to the simple case:

A →β1A’|β2A’| …|βmA’

A’ →α1A’| α2A’| … | αnA’|ε

Example:

exp → exp + term | exp - term |term

remove the left recursion as follows:

exp → term exp’

exp’ → + term exp’ | - term exp’ |ε

**Case 3: General left recursion**

Grammars with no ε-productions and no cycles.

(1) A cycle is a derivation of at least one step the begins and ends with same non-terminal: A=> =>\* A;

(2) Programming language grammars do have ε-productions, but usually in very restricted forms.

Algorithm for general left recursion removal:

for i:=1 to m do

for j:=1 to i-1 do

replace each grammar rule choice of the form Ai→ Ajβ by the rule

Ai→1β|2β| … |kβ, where Aj→1|2| … |k is the current rule for Aj.

remove, if necessary, immediate left recursion involving Ai

Example: consider the following grammar,

A→Ba| Aa| c

B→Bb| Ab| d

Where, A1=A, A2=B and n=2

(1) When i=1, the inner loop does not execute,

So only to remove the immediate left recursion of A

A→BaA’| c A’

A’→aA’| ε

B→Bb| Ab| d

(2) when i=2, the inner loop execute once, with j=1.

To eliminate the rule B→Ab by replacing A with it choices

A→BaA’| c A’

A’→aA’| ε

B→Bb| BaA’b|cA’b| d

(3) remove the immediate left recursion of B to obtain

A→BaA’| c A’

A’→aA’| ε

B→cA’bB’| dB’

B’→bB’ |aA’bB’|ε

Now, the grammar has no left recursion.

**Left Factoring**

Left factoring is required when two or more grammar rule choices share a common prefix string

A→ αβ |αγ => A→ αA’，A’→β|γ

Example

Stmt-sequence→stmt; stmt-sequence | stmt

Stmt→s

Left Factored as follows:

Stmt-sequence→stmt stmt-seq’

Stmt-seq’→; stmt-sequence | ε

Algorithm for left factoring a grammar:

while there are changes to the grammar do

for each non-terminal A do

Letαbe a prefix of maximal length that is shared

By two or more production choices for A

If α≠ ε then

Let A →1|2|…|n be all the production choices for A

And suppose that 1, 2,…, k share , so that A→β1|β2|…|βk| K+1|…|n, the βj’s share no common prefix, andαK+1,…αn do not share 

A →αA’|αK+1|…|αn

A’ →β1|β2|…|βk

**FIRST and FOLLOW SET**

**First Sets**

**Definition:**

Let X be a grammar symbol( a terminal or non-terminal) or ε. Then First(X) is a set of terminals or ε, which is defined as follows:

1. If X is a terminal or ε, then First(X) = {X};

2. If X is a non-terminal, then for each production choice X→X1 X2 … Xn, First(X) contains First(X1)-{ε}.

Let = X1X2…Xn be a string of terminals and non-terminals,. First() is defined as follows:

1.First(α) contains First(X1)-{ε};

2.For each i=2,…,n, if for all k=1,..,i-1, First(Xk) contains ε, then First(α) constains First(Xk)-{ε}.

3. If all the set First(X1)..First(Xn) contain ε, the First(α) contains ε.

**Nullable**

**Definition:**

A non-terminal A is nullable if there exists a derivation A=>\*ε.

**Theorem:**

A non-terminal A is nullable if and only if First(A) contains ε.

**FOLLOW SETS**

Definition:

Given a non-terminal A, the set Follow(A) is defined as follows.

if A is the **start** symbol, the $ is in the Follow(A).

if there is a production B→αAγ, then First(γ)-{ε} is in Follow(A).

if there is a production B→αAγ, such that ε in First(γ), then Follow(A) contains Follow(B).

**Note：**symbol $ is used to mark the end of the input, the empty ε is never an element of a follow set, Follow are defined only for non-terminal

**Example:**

(1) exp → exp addop term

(2) exp → term

(3) addop → + (4) addop → -

(5) term → term mulop factor

(6) term → factor (7) mulop →\* (8) factor →(exp ) (9) factor →number

**The First Sets:**

First(exp)={(,number}

First(term)={(,number}

First(factor)={(,number}

First(addop)={+,-}

First(mulop)={\*}

**The Follow Sets:**

Follow (exp)={ $,+,-,) }

Follow (addop)={(,number}

Follow (term)={$,+,-,\*,)}

Follow (mulop)={(,number}

Follow (factor)={$,+,-,\*,)}

**Constructing LL(1) Parsing Tables**

**Theorem:**

A grammar in BNF is LL(1) if the following conditions are satisfied.

1. For every production A→α1|α2|…|αn, First(αi)∩ First(αj) is empty for all i and j, 1≦i,j≦n, i≠j.

2. For every non-terminal A such that First(A) contains ε, First(A) ∩Follow(A) is empty.

**也就是说LL(1) table其实就是在把所有no-terminal符号的first和follow都加进去，如果没有交集，也就是说表每项最多一个，就是LL(1)**

**Error recovery应该不考？**

**Buttom-Up Parsing**

**Parsing actions:** a sequence of **shift** and **reduce** operations

**Parse State:** a stack of terminals and non-terminals(grows to the right)

**Current derivation step:** always stack + input

**Shift:** shift a terminal from the front of the input to the top of the stack

**Reduce:** Reduce a stringαat the top of the stack to a non-terminal A, given BNF choice -> α

One further feature of bottom-up parsers： grammars are always augmented with a new start symbol.

if S is the start symbol, a new start symbol S' is added to the grammar : S' → S

**Select** *σp*(*r*) = {*t* | *t* ∈ *r* and *p(t)*}

**Project** ∏A1, A2, …, *Ak* (*r*)

**Union** *r* ∪ *s* = {*t* | *t* ∈ *r* or *t* ∈ *s*}

**Set difference** *r – s* = {*t* | *t* ∈ *r* and t ∉ *s*}

**Cartesian product** *r* x *s* = {*t q* | *t* ∈ *r* and *q* ∈ *s*}

**Rename** *ρx[*(*A1, A2, …, An*)] (*E*)

**Set-Intersection** *r* ∩ *s* ={ *t* | *t* ∈ *r* and *t* ∈ *s* }

**Natural-Join** r s

**Theta join** r θs= σθ(r x s)

**Division**

*r* ÷ *s* = { *t* | *t* ∈ ∏ *R-S*(*r*) ∧ ∀ *u* ∈ *s* ( *tu* ∈ *r* ) }

*q = r* ÷ *s,* *q* is the largest rel. satisfying *q* x *s* ⊆ *r*

**Assignment** *temp*1 ← ∏*R-S* (*r*)

**Chapter 3: SQL**

**Data Definition**

**Domain Types in SQL**

