**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

**LR(0) DFA figure**

**The LR(0) Parsing Algorithm**

**Definition**

Let s be the current state (at the top of the parsing stack).Then actions are defined as follows:

1. If state s contains any item of the form A → α·Xβ (X is a terminal). Then the action is to shift the current input token on to the stack.

2. If state s contains any complete item (an item of the form

A → γ·), then the action is to reduce by the rule A → γ·

**A reduction** by the rule S’→ S, where S’ is the start state,

**Acceptance** if the input is empty

**Error** if the input is not empty.

**A grammar is said to be LR(0) grammar if the above rules are unambiguous.**

**A grammar is LR(0) if and only if**

**Each state is a shift state(a state containing only “shift” items)**

**A reduce state containing a single complete item.**

也就是说不能有移进-规约冲突(shift-reduce conflict)或规约-规约(reduce-reduce conflict)冲突。若一个项目包含了完整项目A->α.，那么它就不能再包含其他项目了。实际上，若这样的状态还包含了一个移进项目A->α.Xβ，就会出现一个到底是执行动作1还是动作2的二义性。这种情况称为移进-规约冲突。类似的，如果这样的状态包含了另一个完整项目B->β，那么也会出现一个关于为该规约使用哪个产生式(A->α或B->β)二义性。这种情况称作规约-规约冲突。所以，当晋档每个状态都是移进状态（仅包含了移进项目的状态）或包含了单个完整项目的规约时，该文法才是LR(0)。几乎所有“真正的”文法都不是LR(0)

**LR(0)的图 need tobe appended**

**SLR(1)分析算法**

SLR = Simple LR 简单LR分析，通过使用输入串中的下一个记号来指导它的动作，大大提高了LR(0)分析的能力

**Algorithm 算法：**

1.若状态s包含了格式A->α.Xβ的任意项目，其中X是一个终结符，而且X是输入串中的下一个几号，则动作将当前的输入记号移进栈中，且被压入到栈中的新状态是包含了项目A->αX.β的状态

2.若状态s包含了完整项目A->γ.，则输入串中的下一个几号是在Follow(A)中，所以动作使用规则A->γ规约。用规则S’->S规约与接受等价，其中S是开始状态，只有当下一个输入符号是$时，这才会发生。在所有的其他情况中，新状态都是如下计算的:删除串γ和所有它的来自分析栈中的对应状态。相对应地，DFA回到γ开始构造的状态。通过构造，这个状态必须包括格式B->α,Aβ的一个项目。将A压入栈中，并将包含了项目B->αA.β的状态压入。

3. 若下一个输入几号都不是上面两种情况所提到的，则声明一个错误

A grammar is SLR(1) if and only if, for any state s, the following two conditions are satisfied:

1.For any item A → α·Xβ in s with X a terminal, there is no complete item B → γ. in s with X in Follow(B).

2.For any two complete items A → α·and B →β· in s, Follow(A) ∩ Follow(B) is empty.

Two kinds of parsing conflicts in SLR( l ) parsing

shift-reduce conflicts

reduce-reduce conflicts.

In the case of shift-reduce conflicts, there is a **natural disambiguating rule**: always prefer the shift over the reduce.

The case of reduce-reduce conflicts is more difficult Such conflicts often (but not always) indicate an error in the design of the grammar.

SLR的困难在于它在LR0项的DFA构造之后提供先行，而构造却又忽略了先行。

**LR(1)分析算法**

Write LR(1) items using square brackets as

[A → α.β, a] LR(1)的1就是带上了规约或者移进的条件

where A → αβ is an LR(0) item and a is a token ( lookahead).

The major difference between the LR(0) and LR(1) automata is Definition of the ε-transitions.

The start symbol of the NFA of LR(1) items becomes the item [S’ -> .S, $]

**Parsing algorithm:**

The General LR(1) parsing algorithm :

Let s be the current state (a the top of the parsing stack). Then actions are defined as follows:

1. If state s: any LR(l ) item of the form [A→α·Xβ,a], X is a terminal, and X is the next token in the input string

2. If state s : the complete LR(1) item [A→α·,a] , the next token: in the input string is a

3. If the next input token is such that neither of the above two cases applies, an error is declared.

**A grammar is an LR(1) grammar** :

If the application of the above general LR( l ) parsing rules results in no ambiguity.

**A grammar is LR(1) if and only if,**

for any state s. the following two conditions are satisfied.

l. For any item [A→α·Xβ,a] in s with X a terminal, there is no item in s of the form [B→γ·,X] (otherwise there is a shift-reduce conflict).

2. There are no two items in s of the form [A→α·, a] and [B→β·,a] (otherwise, there is a reduce-reduce conflict).

**LALR(1): Parsing**

These two principles allow us to construct the DFA of LALR(l) items

Identifying all states that have the same core

Forming the union of the lookahead symbols for each LR(0) item.

**Some features**

1.If a grammar is SLR(l), then it certainly is LALR(l)

2. LALR(1) parsers often do as well as general LR(1) parsers in removing typical conflicts that occur in SLR(l) parsing.

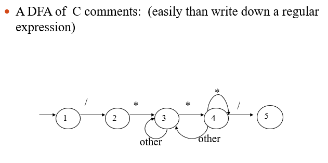
3. If the grammar is already LALR( l ), the only consequence of using LALR( l ) parsing over general LR parsing as following. in the presence of errors, Some spurious reductions may be made before error is declared.

4. Compute the DFA of LALR( l) items directly from the DFA of LR(0) items through a process of propagating lookaheads.

**Chapter 6: Semantic analysis**

**Directed acyclic graphs(DAG) 非循环图**

1.Algorithm must compute the attribute at each node in the dependency graph before it attempts to compute any successor attributes.

2.A traversal order of the dependency graph that obeys this restriction is called a topological sort. 3.The graph must be acyclic(无环的)

**Synthesized and inherited attributes:**

**Synthesized attributes:** An attributes is synthesized: 1. If all its dependencies point from child to parent in the parse tree. 2. Given a grammar A -> X1X2..Xn, the only associated attribute equation with an a on the left-hand side is of the form A.a = f(X1.a1, ….X1.ak,….Xn.a1,…Xn.ak)

If all the attributes are synthesized, then it’s called S-attributed grammar.

Be computed in post-order, often be treated as returned values of the call

**Inherited attributes**: if an attribute is not synthesized, then it’s inherited.

Be computed in pre-order, often be treated as parameters of the call

**L-attributed Definition**:在Xi处aj的值只依赖于在文法规则中Xi左边出现的符号X0,….Xi-1的值。S属性文法是L属性文法

Given an L-attributed grammar in which the inherited attributes do not depend on the *synthesized* attributes:

1、**Top-down parser**: a recursive-descent parser can evaluate all the attributes by turning the *inherited* attributes into *parameters* and *synthesized* attributes into returned values.

2、**Bottom-up parser**: LR parsers are suited to handling primarily synthesized attributes, but are difficult for inherited attributes.

Computing synthesized attributes during LR parsing.

Value stack: store synthesized attributes, be manipulated in parallel with the parsing stack.

**The Symbol table:**

**Structure**

1 Linear List

Provide easy and direct implementation of three basic operations:

Insert operation is performed in constant time.

Lookup and delete operation are linear time in the size of the list.

Good for a compiler implementation in which speed is not a major concern

2 Various Search Tree Structures

(binary, AVL, B trees)

Don’t provide best case efficiency,

the delete operation is very complexity

less useful

3 Hash tables

All three operation can be performed in almost constant time, most frequently in practice, best choice

**Collision resolution**:

1. Open addressing

inserting the collided new items in successive buckets.

cause a significant degradation in performance and make delete operation difficult.

2.  Separate chaining

each bucket is a linear list , collisions are resolved by inserting the new item into the bucket list.

it is the best scheme for compiler construction.

**The Size of the hash table**:

The actual size of the bucket array should be chosen to be a **prime number**

**The algorithm of the Hash function:**

Repeatedly use a constant number α as a multiplying factor when adding in the value of the next character

hi+1 = αhi+ci, h0 = 0

final has value h = hn mod size

h = {Sigma(i=1->n) ci \* α^(n-i) }mod size

**The choice of αhas a significant effect on the outcome, a reasonable choice for α is a power of 2, such as 16 or 128, 因为这样乘法可以用移位来完成**

**Sequential declaration**:

each declaration is added to the symbol table as it is processed.

**Collateral declaration**:

declarations not be added immediately to the existing symbol table

accumulated in a new table(or temporary structure)

added to the existing table after all declarations have been processed.

**Recursive declaration**:

declaration may refer to themselves or each other.

**Chapter 7: Runtime Environment**

General organization of runtime storage

|Code area |

|Global/ static area|

|Stack |

| ↓ |

| Free Space |

| ↑ |

| Heap |

**Fully Static Runtime Environment**

All data are static, remaining fixed in memory for the duration of program execution.

Used for a language, such as FORTRAN77.

no pointer or dynamic allocation.

no recursive procedure calling.

The global variables and all variables are allocated statically.

Each procedure has only a single activation record.

All variables ( local or global) can be accessed directly via fixed address.

No extra information about the environment needs to be kept in an activation record.

**The calling sequence (simple)**:

Each argument is computed and stored into its appropriate parameter location in the activation of the procedure being called.

The return address in the code of the caller is saved.

A jump is made to the beginning of the code of the called procedure.

On return, a simple jump is made to the return address.

1. an extra dereference is required to access parameter values.

2. array parameters do not need to be reallocated and copied.

3. constant arguments must be stored to a memory location and this location used during the call.

The unnamed location is used to store temporary value during the computation of arithmetic expression.

**Stack-based runtime environment**

Recursive calls are allowed

Local variables are newly allocated at each call

Activation records cannot be allocated statically. Instead, activation records must be allocated in a stack-based fashion.

The stack of activation records grows and shrinks with the main of calls in the executing program.

Each procedure may have several different activation records on the call stack at one time.

More complex strategy for bookkeeping and variable access.

Runtime 图

**The calling sequence**:

(1) Compute the arguments and store them in their correct positions in the new activation record of the procedure.

(2) Store the fp as the control link in the new activation record;

(3) Change the fp so that it points to the beginning of the new activation record;

(4) Store the return address in the new activation record;

(5) Perform a jump to the code of the procedure to be called.

**access link** represents the defining environment of the procedure; access link is sometimes also called the static link.

**control link** represents the calling environment of the procedure.

Compiler13 page28