Introduction

编译器定义：Translate on language to another, source language (input) to target language (output). Source language: high-level language, c or c++. Target language: object code, machine code (machine language).

编译过程：(Source code) scanner (tokens) parser (syntax tree) semantic analyzer (annotated tree) source code optimizer (intermediate code) code generator (target code) target code optimizer (target code)

中间过程表格：Literal table, Symbol table, Error Handler

Analysis: lexical analysis, syntax analysis, semantic analysis (optimization).

Synthesis: Code generation (optimization)

Front end: the scanner, parser, semantic analyzer, intermediate code synthesis. From source code to intermediate code.

Back end: the code generator, some optimization.

Passes: process the entire source program several times. a pass consist of several phases

Lexical Analysis

Regular Expression

Describe programming language tokens using regular expressions! A Regular Expression (RE) is defined inductively:

1. a ordinary character stands for itself

2. ε the empty string

3. R|S either R or S (**alternation**), where R, S = RE

4. RS R followed by S (**concatenation**), where R, S = RE

5. R* concatenation of a RE R zero or more times (R* = ε|R|RR|RRR|RRRR..., **repetition**)

优先级：Alternation < Concatenation < Repetition

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

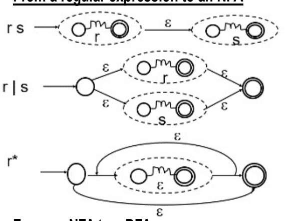
C: (“/*”) (“/”)* ([^*/] | [^*] “/” | “*” [^/])* (“*”)* (“*/”)

Pascal: { [^}]* }

Schema: ; [^newline]*

Ada: -- [^newline]*

From a regular expression to an NFA



From an NFA to a DFA

1.the ε-closure of a Set of states:

the ε-closure of a single state s is the set of states reachable by a series of zero or more ε-transitions. The ε-closure of a set of states : the union of the ε-closures of each individual state.

2. the Subset Construction:

1) Compute the ε-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.

a) 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.

b) 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 states).

c) 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.

d) 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.

e) If 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.

Context-Free Grammars and Parsing

Parse Trees

A left-most 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：Appear "higher" (that is, closer to the root) in the parse and syntax trees, and thus receive lower precedence.

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 left recursive rule makes its operators associate on the left. A right recursive rule makes them associate on the right.

利用左递归来实现左结合，右递归来实现右结合

The dangling else problem 悬挂 else 问题

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**

In C:

statement → *matched-stmt* | *unmatched-stmt*
matched-stmt → **if** (*exp*) *matched-stmt* **else** *matched-stmt* | **other**
unmatched-stmt → **if** (*exp*) *statement* | **if** (*exp*) *matched-stmt* **else** *unmatched-stmt*
exp → **0** | **1**

In Ada:

if-stmt → **if** condition **then** statement-sequence **end if** | **if** condition **then** statement-sequence **else** statement-sequence **end if**

Inessential ambiguity 无关紧要的二义性

Sometimes a grammar may be ambiguous and yet always produce unique abstract syntax trees.

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

$A \to A\,\alpha \mid \beta$ (left recursive)
$A \to \alpha\, A \mid \beta$ (right recursive)

EBNF opts to use curly brackets {. . .} to express repetition.

$A \to \beta\,\{\alpha\}$ (left recursive)
$A \to \{\alpha\}\,\beta$ (right recursive)

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:

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

2) 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 $);

3) Any entrances remaining empty represent potential errors.

Left Recursion Removal and Left Factoring

Case1: Simple immediate left recursion 直接左递归

A → A α | β

Rewrite this grammar rule into two rules;

A→ βA'   A'→ αA' | ε   Example is as

exp → exp addop term | term
exp → term exp'
exp' → addop term exp' | ε

Case 2: General immediate left recursion 一般形式

$A \to A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_m$

Where none of $\beta_1,\dots,\beta_m$ begin with A.

The solution is similar to the simple case:

$A \to \beta_1A' \mid \beta_2A' \mid \dots \mid \beta_mA'$
$A' \to \alpha_1A' \mid \alpha_2A' \mid \dots \mid \alpha_nA' \mid \varepsilon$

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 that 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:

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→ α β | α γ => A→α A', 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**

FIRST and FOLLOW SET

First Sets

Definition:

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

If X is a non-terminal, and let X → $X_1X_2\dots X_n$ be a string of terminals and non-terminals,.

First(X) is defined as follows:

1) First(X) contains First(X_1) - {ε};

2) For each i=2,...,n, if for all k=1,...,i-1, First(X_k) contains ε, then First(X) contains First(X_k)-{ε}.

3) If all the set First(X_1)...First(X_n) contain ε, the First(X) 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.

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

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

3) 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 $A \to \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$, First($\alpha_i$) ∩ First($\alpha_j$) 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 表中每项最多一个，就是 LL(1)

The LL(1) parsing **table-constructing rules**:

1) For each token a in First(α), add A→α to the entry M[A,a].

2) If ε is in First(α), for each element a of Follow(A) (a token or $), add A→α to M[A,a].

Bottom-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 string α at 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时，该文法才是 LR(0)。几乎所有“真正的”文法都不是 LR(0)

SLR(1) Parsing 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记号，则动作将当前的输入记号移进栈中，且被移进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记号是在 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•Xβ的一个项目。将 A 压入栈中，并将包含了项目 B->α•Xβ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(1) 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) Parsing 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.

1) 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).

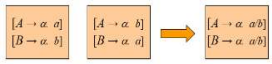
LALR(1) Parsing 分析算法:

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.

may be **reduce-reduce conflicts**.



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.

Semantic analysis

Directed acyclic graphs(DAG) 非循环图

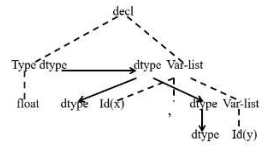
1) Algorithm must compute the attribute at each node in the dependency graph before it attempts to compute any successor attributes.

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)

Dependency Graphs

decl → *type var-list*
type → *int* | *float*
var-list → *id,var-list* | *id*



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.

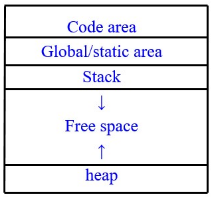
The Size of the hash table
The actual size of the bucket array should be chosen to be a **prime number**

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
$h_{i+1} = \alpha h_i + c_i$, $h_0 = 0$
final has value h = hn mod size
h = {Sigma(i=1->n) c_i *α^(n-i)}mod size
The choice of α has 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.

Runtime Environments

General organization of runtime storage



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.
1) The return address in the code of the caller is saved.
2) A jump is made to the beginning of the code of the called procedure.
3) On return, a simple jump is made to the return address.
In FORTRAN77:
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
1) Recursive calls are allowed
2) Local variables are newly allocated at each call
3) Activation records cannot be allocated statically. Instead, activation records must be allocated in a stack-based fashion.
4) The stack of activation records grows and shrinks with the main of calls in the executing program.
5) Each procedure may have several different activation records on the call stack at one time.
6) More complex strategy for bookkeeping and variable access

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.
可以通过 Control Link 来找到每个函数中的局部变量，但是必须在执行时保存用于每个过程的局部符号表，这样才能允许在每个活动记录中查询标识符，以及若它退出的话也可以看到，并且可以判定出它的偏移。这是运行时环境的最主要额外复杂性。解决这个问题的方法也实现静态作用域，是将一个 access link 额外信息添加到每个活动记录中，除了可以指向代表过程的定义环境而不是调用环境之外，访问链与控制链相似，正是由于这个原因，即使它不是编译时决定的量，访问链有时也被成为静态链(static link)

Heap Management:
A standard method for maintaining the heap and implementing these functions:
(1) A circular linked list of free blocks.
(2) Memory is taken by malloc.
(3) Memory is return by free.
Drawbacks:
(1) The free operation can not tell if the pointer is legal or not.
(2) Care must be taken to coalesce blocks, otherwise, the heap can quickly become fragmented.
A different implementation of malloc and free.
use a circular linked list data structure that keep track of both allocated and free block.

Automatic Management of the heap:
Mark and sweep garbage collection
No memory is freed until a call to malloc fails, which does this in two passes.
(1) Follows all pointers recursively, starting with all currently accessible pointer values and marks each block of storage reached.
(2) Sweeps linearly through memory.
returning unmarked blocks to free memory.
perform memory compaction to leave only one large block of contiguous free space at the other end.
Drawbacks:1. Require extra storage 2. The double pass through memory cause a significant delay in processing
这种方法中，直到一个对 malloc 的调用失败之前都不会释放存储器，在这是将垃圾回收程序激活，寻找可被引用的所有存储器并释放所有未引用的存储器。这是通过两遍来完成的。第一遍递归的顺着所有的指针前进，从所有当前的可访问指针值开始，并标出到达的每个存储器块。这个过程要求额外的位存储标识，另一个则线性地打扫存储器，并将未标出的块反馈到自由存储器中。虽然这个过程通常要寻找足够的相邻自由存储器以满足一系列的新要求，但存储器有可能非常破碎，故尽管是在垃圾回收之后，大的存储请求仍旧会失败，因此，垃圾回收经常也会通过将所有的分配的空间移到堆的末尾，以及在另一端留下相邻的自由空间的唯一一个大型块儿执行存储器压缩(memory compaction)。这个过程还必须在存储器中更新对那些在执行程序时被移掉的区域的所有引用。标识和打扫垃圾回收有若干个缺点：它要求额外的存储（用于标识），在存储气中的两个遍历导致了过程中很大的延迟，有时需要几秒钟，而每一次的调用垃圾回收程序又都需要几分钟时间。这对于那些许多设计到了交互和即时相应的应用程序显然是不合适的。

Stop-and-copy or two-space garbage collection
(1) During the marking pass, all reached blocks are immediately copied to the second half of storage not in use;
(2) No extra mark bit is required and only one pass is required;
(3) It also performs compaction automatically.
(4) It does little to improve processing delays during storage reclamation.
可以通过将可用的存储器分为两个部分并每次只从一个部分中分配存储来对这个过程进行改进。在标识遍历时，将所有达到了的块都复制到未被使用的另一半存储器中，这就意味着在存储式不在要求额外的标志位而且只需要一遍就够了。它还自动地进行压缩。一旦位于使用的区域中的所有可到达的块都复制好时，就将使用的和未使用的存储器部分相互交换，而过程依然继续进行。对存储回收过程中的过程延迟改进不大。

General garbage collection:
Allocated objects that survive long enough are simply copied into permanent space and are never deallocated during subsequent storage reclamations.
这就意味着垃圾回收程序在更新的存储分配时只需要搜搜存储器中很小的一部分，当然永久存储器也有可能由于不可达到的存储而用尽，但这相对于前面的问题就不那么严重了，这是因为临时存储会很快消失，而可被分配的存储则总会有的。人们已经证明了这个存储方法很好，在虚拟存储系统中尤为如此。

Parameter Passing Mechanisms:
Pass by value
The arguments are expressions that are evaluated at the time of the call.
Their values becomes the values of the parameter during the execution of the procedure.
The only parameter passing mechanism available in C; C 常用的方式
The default in Pascal and Ada
Pass by reference
Pass by reference passes the location of the variable.
The parameter becomes an alias for the argument.
The only parameter passing mechanism in Fortran77
In Pascal, pass by reference achieved with the use of var keyword
In C++, by the use of special symbol & in the parameter declaration
在引用传递中，自变量必须与分配的地址一起变化。并非传递变量的值，引用传递的是变量的地址，因此参数就变成了自变量的别名(alias)，而且在参数上发生的任何变化都会出现在自变量上
Fortran 这样只可用引用传递的语言中，要为不带地址的值的自变量提供一个位置如 P(2+3)是非法的，编译程序必须为 2+3 创造一个地址并把 2+3 存进去在进行引用
Pass by value-result
The mechanism achieves a similar result to pass by reference, except that no actual alias is established.
Known as copy-in, copy-out, or copy- restore.
The value of the argument is copied and used in the procedure.
The final value of the parameter is copied back out to the location of the argument.
This is the mechanism of Ada in out parameter.
在过程中复制和使用自变量的值，当过程退出时，再将参数的最终值复制回自变量的地址。

```
void p(int x, int y)
{ ++x;
  ++y;
}
main()
{ int a=1;
  p(a, a);
return 0;
}
```

If pass by reference, a is 3, if pass by value-result, a is 2
Pass by name
This is the most complex of the parameter passing mechanisms. (delayed evaluation)
Idea:
The argument is not evaluated until its actual use in the called program.
思想是知道在被调用的程序真正使用了自变量（作为一个参数）之后才对这个自变量赋值，所以它还被称为延迟赋值(delayed evaluation)因此，自变量的名称或是它在调用点上的结构表示取代了它对应的参数的名字。

```
int i;
int a[10];

void p(int x)
{++i;
++x;
}
```

```
main()
{
 i=1;
a[1]=1;
a[2]=2;
p(a[i]);
return 0;
}
```

对 p 的调用的结果是将 a[2]设置为 3 并保持 a[1]不变
在调用点上的自变量的文本被看成是它自己右边的函数，每当在被调用的过程的代码中到达相应的参数名时，就要计算它。

Code Generation

Three-Address Code:
The most basic instruction of three address code is designed to represent the evaluation of arithmetic expressions and has the following general form:
x = y op z
其中 x 的地址必须不同于 y,z 的地址，y,z 可以代表常量，但是 x 不行
2*a+(b-3)转换为
T1 = 2 * a
T2 = b – 3
T3 = T1 + T2
其中 T1, T2, T3 均为临时变量
Data Structures for the Implementation of Three-Address Code:
1) Four fields are necessary: one for the operation and three for the addresses. Such a representation of three-address code is called a quadruple.
2) Those instructions that need fewer than three addresses, one or more of the address fields is given a null or "empty" value.
3) The entire sequence of three-address instructions is implemented as an array or linked list.
4) We allow an address to be only an integer constant or a string (representing the name of a temporary or a variable).
5) Since names are used, these names must be entered into a symbol table, and lookups will need to be performed during further processing.
6) An alternative to keeping names in the quadruples is to keep pointers to symbol table entries. This avoids the need for additional lookups and is particularly advantageous in a language with nested scopes.
7) A different implementation of three-address code is use the instructions themselves to represent the temporaries. Such an implementation of three-address code is called a triple.
P-Code:
P-code began as a standard target assembly code produced by a number of Pascal compilers of the 1970s and early 1980s.
It was designed to be the actual code for a hypothetical stack machine, called the P-machine, for which an interpreter was written on various actual machines.
The P-machine consists of a code memory, an unspecified data memory for named variables, and a stack for temporary data, together with whatever registers are needed to maintain the stack and support execution.
Example:
2*a+(b-3):
ldc 2 ;load constant 2 (pushes 2 onto the temporary)
lod a ;load value of variable a(pushes a onto the temporary)
mpi ;integer multiplication (pops these two values from the stack, multiplies them (in reverse order), and pushes the result onto the stack.)
lod b ;load value of variable b
ldc 3 ;load constant 3
sbi ;integer subtraction(subtracts the first from the second)
adi ;integer addition
x:=y+1:
lda x ;load address of x
lod y ;load value of y
ldc 1 ;load constant 1
adi ;add
sto ;store top to address below top & pop both
other:
stn : stores the value to the address but leaves the value at the top of the stack, while discarding the address.
1) P-code is in many respects closer to actual machine code than three-address code. P-code instructions also require fewer addresses:
2) P-code is less compact than three-address code in terms of numbers of instructions, and P-code is not " self-contained " in that the instructions operate implicitly on a stack.
3) Historically, P-code has largely been generated as a text file, but the previous descriptions of internal data structure implementations for three-address code will also work with appropriate modification for P-code.
Code Generation
Intermediate code generation can be viewed as an attribute computation. This code becomes a synthesized attribute that can be defined using an attribute grammar and generated either directly during parsing or by a postorder traversal of the syntax tree.

Begin
If T is not nil then
Generate code to prepare for code of left child of T;
Gencode(left child of T);
Generate code to prepare for code of right child of T;
Gencode(right child of T);
Generate code to implement the action of T;
End;

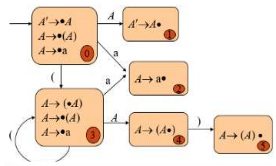
Data Structure References
Three-Address Code:
t1 = &x +10
*t1 = 2
P-Code:
ind (" indirect load") ind i
ixa ("indexed address")
Each address must be computed from the base address of a (its starting address in memory) and an offset that depends linearly on the value of the subscript.
The offset is computed from the subscript value as follows.
1) An adjustment must be made to the subscript value if the subscript range does not begin at 0.
2) The adjusted subscript value must be multiplied by a scale factor that is equal to the size of each array element in memory. Finally, the resulting scaled subscript is added to the base address to get the final address of the array element.
Example:
a[i+1] = a [j*2]+3
Three-Address Code:
t1 = j * 2 t2 = a [t1] t3 = t2 + 3
t4 = i + 1 a [t4] = t3
Three-Address Code:
t1 = j * 2 t2 = t1 * elem_size(a) t3 = &a + t2
t4 = *t3 t5 = t4 + 3 t6 = i + 1
t7 = t6 * elem_size (a) t8 = &a + t7 *t8 = t5
P-Code:
lda a lod i ldc 1 adi ixa elem_size(a)
lda a lod j ldc 2 mpi ixa elem_size(a)
ind 0 ldc 3 adi sto
Control Statement:
False Jump: if_f, fjg
Unconditional Jump: goto ujp
Three-Address Code:
if (E) S1 else S2
<code to evaluate E to t1>
if_false t1 goto L1
<code for S1>
goto L2
label L1
<code for S2>
label L2
while (E) S
label L1
<code to evaluate E to t1>
if_false t1 goto L2
<code for S>
goto L1
label L2
P-Code:
if (E) S1 else S2
<code to evaluate E>
fjp L1
<code for S 1>
ujp L2
lab L1
<code for S 2>
lab L2
while (E) S
lab L1
<code to evaluate E>
fjp L2
<code for S>
ujp L1
lab L2
Logical Expressions
(x != 0) && (y == x)
lod x ldc 0 neq fjp L1 lod y lod x equ fjp L1 ujp L2 lab L1 lod FALSE lab L2
Procedure and Function Calls
int f (int x, int y)
{ return x + y + 1; }
Three-Address Code:
entry f t1 = x + y t2 = t1 + 1 return r2
P-Code:
ent f lod x lod y adi ldc 1 adi ret
The call **f(2+3, 4)**
Three-Address Code:
begin_args t1 = 2 + 3 arg t1 arg 4 call f
P-Code:
mst: mark stack, the same as begin_args in Three-Address Code
cup: call user procedure
mst ldc 2 ldc 3 adi ldc 4 cup f
Example:
fn f(x) = 2+x;
fn g(x,y) = f(x) +y;
g(3,4)
P-Code:
ent f ldc 2 lod x adi ret
ent g mst lod x cup f lod y adi ret
mst ldc 3 ldc 4 cup g

More Examples

LL(1):
S → ES'
S' → ε | +S
E → num | (S)

	num	+	(	)	$
S	S→ES'		S→ES'		
S'		S'→+S		S'→ε	S'→ε
E	E→num		E→(S)		

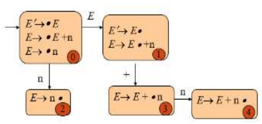
LR(0):
A → (A) | a



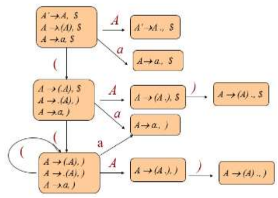
State	Input			Goto
	(	a	)	A
0	s3	s2		1
1	r1	r1	r1	
2	r2	r2	r2	
3	s3	s2		4
4			s5	
5	r3	r3	r3	

	Parsing stack	Input	Action
1	$0	((a))$	Shift
2	$0 (3	(a))$	Shift
3	$0 (3 (3	a))$	Shift
4	$0 (3 (3 a2	))$	Reduce A→a
5	$0 (3 (3 A4	))$	Shift
6	$0 (3 (3 A4)5	)$	Reduce A → (A)
7	$0 (3 A4	)$	Shift
8	$0 (3 A4)5	$	Reduce A → (A)
9	$0 A1	$	Accept

SLR(1):
Not LR(0), but SLR(1)



LR(1):
A → (A) | a



State	Input				Goto
	(	a	)	$	S
0	s2	s3			1
1				accept	
2	s5	s6			4
3				r2	
4			s7		
5	s5	s6			8
6			r2		
7				r1	
8			s9		
9			r1		

LALR(1):
A → (A) | a