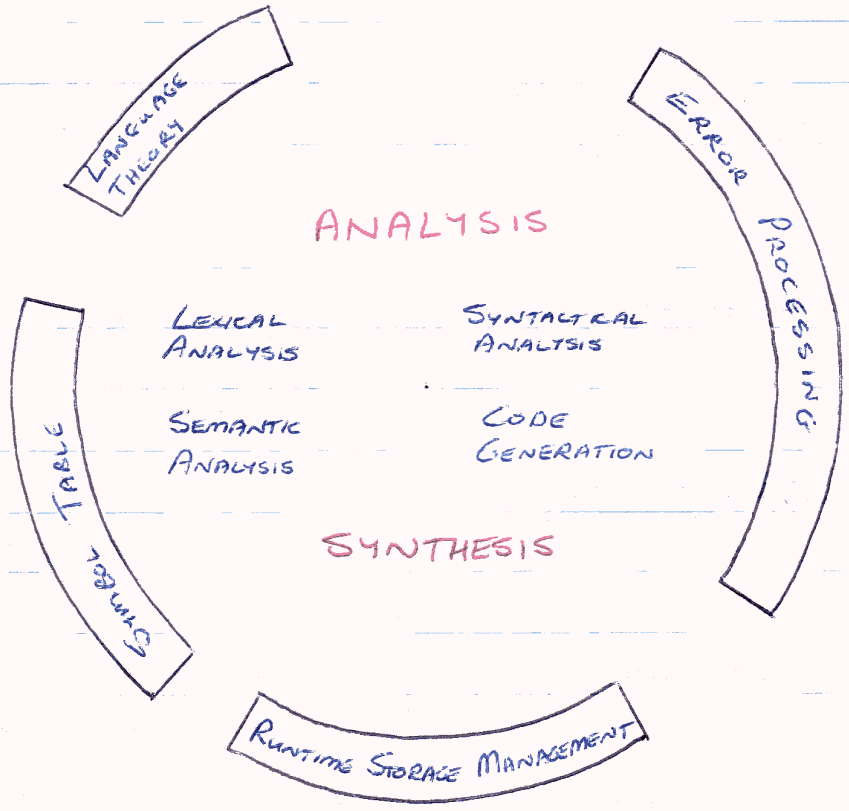
COMPILERS

① A compiler is one type of translator, converting source → object code, where the source is a high level language and the object is a low level language; ie, it transforms progs from the user's domain of problem solving into the machine's
② domain of program execution without varying the semantics.

Any compilation can be broken down into two major tasks:
Analysis: discover the structure and basic symbols of the source program determining its meaning
Synthesis: create a target program equivalent to the source program

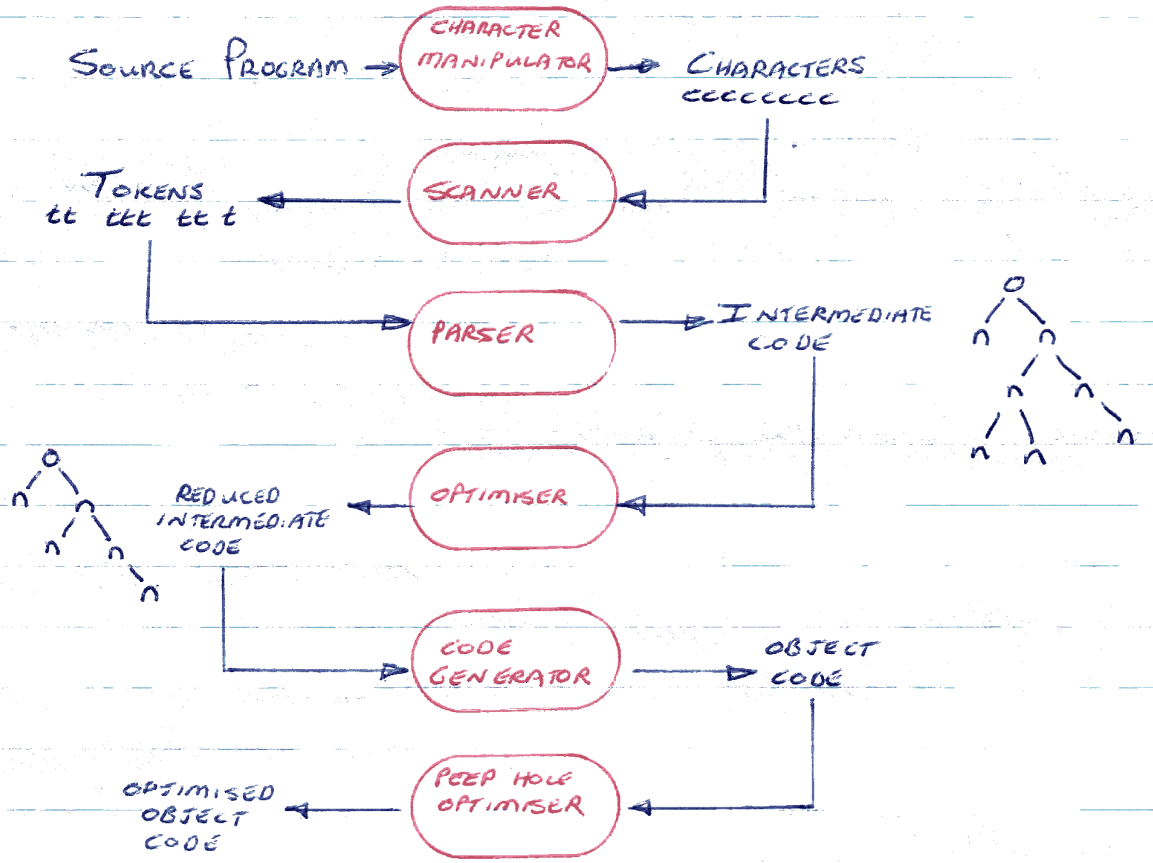


The analysis phase is concerned solely with the source language. It converts the source program into an abstract representation - usually a tree. Synthesis then takes this representation and translates it to target m/c.

(2)

The compiler can be broken up into a number of different phases:

(3)



CHARACTER MANIPULATOR

This takes as input records submitted by the O/S and splits them into characters, taking into account the end of record conditions

(4)

SCANNER (LEXICAL ANALYSER)

This groups the characters into tokens, and removes noise symbols such as comments and blanks

PARSER (SYNTAX ANALYSER)

This accepts the tokens and checks whether they occur in

syntactically valid patterns. The output is an intermediate code which can be in normal language form or in a tree structured form. The latter is more useful as it can be easily manipulated.

OPTIMISER

This (optional) step produces a reduced intermediate code requiring fewer operations

CODE GENERATOR

(5) This transforms the intermediate code into code for the target computer. The code will not be particularly efficient, however.

PEEP HOLE OPTIMISER (optional)

This accepts short sequences of code and determines whether they can be replaced by equivalent shorter code sequences

ERROR RECOVERY

This system is normally attached to the parser, taking control upon error detection. Purpose: diagnose error, attempt to print meaningful error message, and recover so that compilation can continue. Recovery may involve inserting or discarding tokens.

(4)

(6) # Language Theory

We will use set theory to describe our languages. We specify the syntactic rules in a grammar. When specifying a programming language both the syntax (structure, form) and semantics (meaning) must be specified

(7) Defns. An alphabet Σ is a finite set of symbols for the defined source language.

A string or word over Σ is a sequence of symbols from Σ.
A string of length 0 is denoted by ε

A grammar G is a 4-tuple (N, T, P, S)
where: N is a finite set of non-terminal symbols
T is a finite set of terminal symbols : $T \cap N = \emptyset$
S is a unique symbol $(\in N)$ called the start or sentence symbol
P is a set of productions of the form $u \to v$
where $u \in (N \cup T)^+$ and $v \in (N \cup T)^*$

To simplify our productions we write them in EBNF, observing the following:
- each production is terminated by a period
- terminal symbols are usually delimited by apostrophes
- alternatives are indicated by |
(8) - curly brackets indicate 0 or more $(\equiv (\ )^*)$
- square brackets mean 0 or 1 of an item
- round brackets group items together eg $(A|B)C \equiv AC|BC$

(9) Chomsky Grammers

N. Chomsky introduced a notation and classification of grammars by placing restrictions on the type of productions allowed. There are four types in his scheme, each a subset of the previous one:

Type 0 or Unrestricted

This is the most general type, with no restrictions. We will not discuss these further.

Type 1 or Context Sensitive

All productions must be of the form $PAQ \to PBQ$ where P, Q are (possibly empty) strings of $(N \cup T)^*$. A is a single non-terminal and B any non-empty string of $(N \cup T)^+$. Thus we allow transformations of A to B in the context of P and Q; P is the left context, Q the right. These are still too general for compilers.

Type 2 or Context Free (CFG)

All productions have the form $A \to V$ where $A \in N$ and $V \in (N \cup T)^*$

Type 3 or Regular

There are two forms:

Right linear: all productions are $A \to tB$ or $A \to t$

Left linear: all " " $A \to Bt$ or $A \to t$

where $A, B \in N$, $t \in T$

(6)

All computer languages are type 2 although type 3 languages can also be used for parts of language specification, particularly lexical analysis

(10) RASCAL/2 DESCRIPTION (produced by A. B. PYSTER)

PROGRAM → HEADING [CONSTANTS] [VARS] BEGIN-END-STMT .
HEADING → 'PROGRAM' ID ['(' PROG-PARMS ')'] ';' .
CONSTANTS → 'CONST' CONST-STMT {CONST-STMT} .
VARS → 'VAR' VAR-STMT {VAR-STMT} .
BEGIN-END-STMT → 'BEGIN' {EXEC-STMT} 'END' ';' .

CONST-STMT → ID '=' ID ';' | ID '=' LIT ';' .
VAR-STMT → IDS ':' TYPE .
IDS → ID {',' ID}
TYPE → 'INTEGER' | 'BOOLEAN' .
INTEGER → DIGIT {DIGIT} .
BOOLEAN → 'TRUE' | 'FALSE' .
LIT → BOOLEAN | ['+' | '-'] INTEGER
DIGIT → '0' | '1' | ... | '9' .
EXEC-STMT → BEGIN-END-STMT | ASSIGN-STMT | IF-STMT |
WHILE-STMT | REPEAT-STMT | NULL-STMT |
READ-STMT | WRITE-STMT .
ASSIGN-STMT → ID ':=' EXPRESSION ';' .
EXPRESSION → TERM [REL-OP TERM]
TERM → ['+' | '-'] FACTOR {ADD-OP FACTOR}
FACTOR → PRIMARY {MULT-OP PRIMARY}
PRIMARY → INTEGER | ID | '(' EXPRESSION ')' | -['NOT'] BOOLEAN .
REL-OP → '=' | '>' | '<' | '<>' | '<=' | '>=' .
ADD-OP → '+' | '-' | 'OR' .
MULT-OP → '*' | 'DIV' | 'MOD' | 'AND' .

IF-STMT → 'IF' EXPRESSION 'THEN' EXEC-STMT ['ELSE' EXEC-STMT].

WHILE-STMT → 'WHILE' EXPRESSION 'DO' EXEC-STMT.

REPEAT-STMT → 'REPEAT' {EXEC-STMT} 'UNTIL' EXPRESSION ';'.

NULL-STMT → ';'.

PROG-PARMS → '(' 'INPUT' ['.' OUTPUT '.'] ')' | '(' 'OUTPUT' ['.' 'INPUT'] ')'.

READ-STMT → ('READ' | 'READLN') ['(' IOS ')'].

WRITE-STMT → ('WRITE' | 'WRITELN') [WRITE-LIST].

WRITE-LIST → '(' EXPRESSION {',' EXPRESSION} ')'.

ID → LETTER {LETTER | DIGIT}.

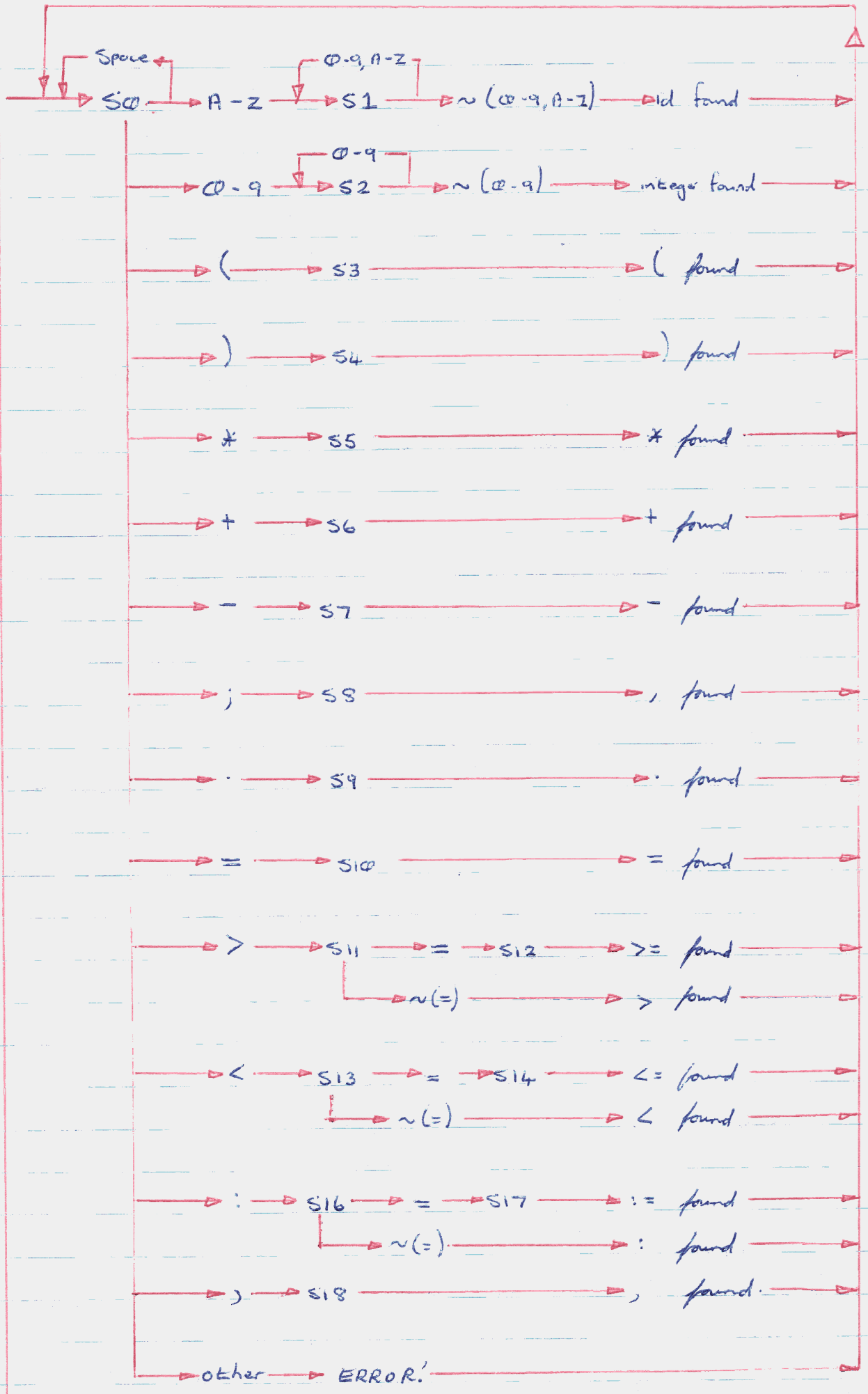
LETTER → 'A' | 'B' | ... | 'Z'.

(12) ## LEXICAL ANALYSIS

This consists of the character manipulator and the scanner.
The purpose of the L.A. is
- to accept the source prog one char at a time and convert it to an internal representation for use by the rest of the compiler in the form of tokens (eg keywords, ids., constants, operators, etc)
- to remove extraneous blanks, carriage returns and comments
- to report any errors discovered.

In our language RASCAL/2, the lexical items are:
- identifiers (beginning with a letter)
- integers (beginning with a digit)
- operators < > >= <= <> + * - :=
- seperators ( ) ; .

We can represent our L.A. by a finite state automaton where the states are represented by Sn and the lines the paths taken by the various chars:


Space
S0
A-Z
0-9, A-Z
S1
~(0-9, A-Z)
Id found
0-9
0-9
S2
~(0-9)
integer found
(
S3
(found
)
S4
) found
*
S5
* found
+
S6
+ found
-
S7
- found
;
S8
; found
.
S9
. found
=
S10
= found
>
S11
=
S12
>= found
~(=)
> found
<
S13
=
S14
<= found
~(=)
< found
:
S16
=
S17
:= found
~(=)
: found
)
S18
) found
other
ERROR!

(14) We can identify from the next character in the input string what state we should enter and whether we have identified a lexical unit or not - a one-character look ahead.

For a finite state automaton to recognise our lexical units, the language specifying the lexical units must be a regular (Type 3) grammar.

Lex an. could be a seperate pass of the compiler, placing its output on an intermediate file, but usually the L.A. forms a subroutine of the parser, called by the parser whenever the next token is needed. The L.A. then returns an integer code for simple constructs, or an integer code and pointer for more complex constructs. The integer code gives the token type, the pointer points to the value of the token (eg constants) or to the symbol table (eg ids)

Reserved words can either be treated as identifiers, with a table lookup performed to determine whether the identifier is a keyword or not, or they can be entered into the symbol table with an indicator that the word is reserved.

(15) The lexical analyser is normally divided into two routines - getchar and gettoken. Getchar supplies one character on each call (ie, is responsible for omitting eolns and possibly comments (NOT spaces)). If the eof is encountered by getchar, a PROGRAM INCOMPLETE error message is generated.

Gettoken corresponds to our finite state automata. It calls getchar from a while loop until a non-space is found (getcha

(10)

should not ignore spaces as spaces may be important in strings, etc.)

(20) Once the lexical analyser is implemented, the compiler need only handle tokens as opposed to characters.

(21) ## GRAMMARS REVISITED

We may either start with a distinguished symbol and apply productions to get a sentence containing only basic symbols, or start with our sentence and work backwards to the start symbol. The former method is called *generative*, the latter *recognitive*. Parsers may use either method - generative parsers are called *top-down parsers* recognitive *bottom-up parsers*.

In order to implement a parser, the syntax of the source language must be specified by means of a context free (type 2) grammar. These are accepted by a FSA controlling a push-down stack with certain simple rules governing its operations.

DEFN A sequence $a_0 \Rightarrow a_1 \Rightarrow \ldots \Rightarrow a_n$ is a *derivation* of a_n from a_0. Sentences of languages are all derivations from the start symbol S.

(22) Consider the grammar $S \to E$

$E \to E+E \mid E*E \mid (E) \mid -E \mid id$

Then a derivation of id * (-id) could be:

$S \Rightarrow E \Rightarrow E*E \Rightarrow id*E \Rightarrow id*(E) \Rightarrow id*(-E) \Rightarrow id*(-id)$

Another derivation is:

$S \Rightarrow E \Rightarrow E*E \Rightarrow E*(E) \Rightarrow E*(-E) \Rightarrow E*(-id) \Rightarrow id*(-id)$

The first derivation is called a <u>leftmost derivation</u>, the latter a <u>rightmost derivation</u> (ie the rightmost non-terminal is replaced at each step). All the intermediate forms between the start symbol and the sentence are known as <u>sentential forms</u>. In the derivation $E*E \Rightarrow E*(E)$, $E*E$ is the <u>left sentential form</u> and $E*(E)$ is the <u>right sentential form</u>.

A sequence of one or more derivation steps is denoted by $\Rightarrow^+$ and zero or more $\Rightarrow^*$. So, for eg, $S \Rightarrow^* S$ and $S \Rightarrow^+ E*E$

(23) ## Parse Trees

We can use our productions to expand each S into a particular (valid) string (ie derive our string) and represent the derivation in a tree structure known as a parse tree. If our string is legal, a complete parse tree exists (a complete parse tree is one in which all leaves are terminals).

It is quite possible that the leftmost derivation and rightmost derivations will result in different parse trees (27) for the same string. A grammar in which this occurs is said to be ambiguous - and is a highly undesirable basis for a programming language, as a statement may have two different meanings. (eg if *, + are given the same hierarchical level in the grammar but not in our

interpretation of a string, we can get two different values)

We can often disambiguate a grammar by specifying the associativity and precedence of the arithmetic operators.

Suppose the precedence of operators is $-$, $*$ /, $+-$ (the first $-$ being the unary minus) We disambiguate the grammar by introducing one non-terminal for each precedence level. An indivisible expression we call a primary. Primaries with zero or more of the highest precedence operators are called factors, and so on.

Eg Consider the grammar:

STATEMENT → EXPRESSION .

EXPRESSION → EXPRESSION '+' EXPRESSION |
EXPRESSION '−' EXPRESSION |
EXPRESSION '*' EXPRESSION |
EXPRESSION '/' EXPRESSION |
'(' EXPRESSION ')' |
'−' EXPRESSION |
a .

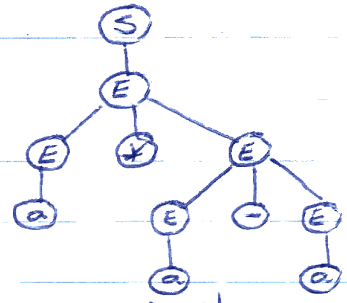
This is ambiguous, eg a * a − a has the following derivations:

Leftmost : $S \Rightarrow E \Rightarrow E*E \Rightarrow a*E \Rightarrow a*E-E \Rightarrow a*a-E \Rightarrow a*a-a$

Rightmost : $S \Rightarrow E \Rightarrow E-E \Rightarrow E-a \Rightarrow E*E-a \Rightarrow E*a-a \Rightarrow a*a-a$

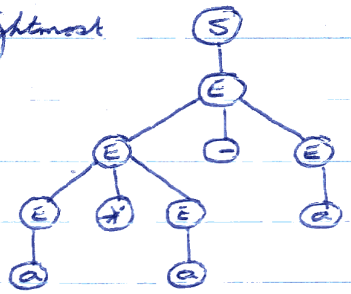
The parse trees are:

Leftmost



WRONG!

Rightmost



RIGHT!

(28) We introduce the productions for terms, factors and primaries, giving us the unambiguous grammar:

STATEMENT → EXPRESSION .

EXPRESSION → EXPRESSION '+' TERM |
EXPRESSION '-' TERM |
TERM .

TERM → TERM '*' FACTOR |
TERM '/' FACTOR |
FACTOR .

FACTOR → '-' FACTOR | PRIMARY .

PRIMARY → '(' EXPRESSION ')' | a .

(29) ## PARSING (SYNTAX ANALYSIS)

We will consider only top-down parsing (as in the previous eg), concentrating initially on recursive descent parsers (parsers where every non-terminal is broken down by recursive calls). This type is easy to write and fast to implement but is not normally used in commercial compilers (except Pascal).

The parse tree generated by this method is in fact figurative; it exists only as a sequence of actions made by

stepping through the tree construction process. At each step the leftmost non-terminal is expanded. If our productions present us with alternatives, we will choose the first and only if this fails choose the second, etc.

Eg Consider the grammar:

STATEMENT → EXPRESSION

EXPRESSION → TERM | EXPRESSION '+' TERM

TERM → FACTOR | TERM '*' FACTOR

FACTOR → '(' EXPRESSION ')' | a

with input string a + a * a we derive:

STATEMENT ⇒ EXPRESSION ⇒ TERM ⇒ FACTOR ⇒ '(' EXPRESSION ')'

(30) We get a terminal symbol '(' which we match against the first character in the input string, a. These do not match, so we try the alternative FACTOR ⇒ a. Although this matches the first character, it is completely terminal, and does not match the rest of the string. There are no more alternatives for FACTOR, so we back up to TERM and try TERM ⇒ TERM '*' FACTOR.

This too is wrong but our parser will never discover this, as each time we backup to term we expand by TERM → TERM '*' FACTOR and never backup higher to EXPRESSION. This is because the grammar is left recursive.

A grammar G is <u>left recursive</u> if it has a non-terminal A such that there is a derivation A ⇒* Ab for some b. A left recursive grammar will cause a top-down parser to loop infinitely

Backtracking is slow and requires the semantic effects of partially parsed expressions to be undone. The order of the alternatives in a production can also affect the language
(31) accepted. The recursive descent parser eliminates the need for backtracking over the input.

ELIMINATION OF LEFT RECURSION.

If we have a left recursive production $A \rightarrow Aa \mid B$
we can replace this by $A \rightarrow B\ A1$
$A1 \rightarrow a\ A1 \mid e$

where B ∈ NUT and B does not begin with A.
Our previous grammar could become:

STATEMENT → EXPRESSION.
EXPRESSION → TERM REST-OF-EXPR.
REST-OF-EXPR → '+' TERM REST-OF-EXPR | e
TERM → FACTOR REST-OF-TERM.
REST-OF-TERM → '*' FACTOR REST-OF-TERM | e
FACTOR → '(' EXPRESSION ')' | a

In general to eliminate direct left recursion among all the productions of a non-terminal A we group these productions as:

$A \rightarrow A\,a1 \mid A\,a2 \mid \ldots \mid A\,am \mid B1 \mid B2 \mid \ldots \mid Bn$.

where no Bi begins with an A. We then replace the A productions by two productions:

$A \rightarrow B1\ A1 \mid B2\ A1 \mid \ldots \mid Bn\ A1$.
$A1 \rightarrow a1\ A1 \mid a2\ A1 \mid \ldots \mid am\ A1 \mid e$.

To eliminate indirect left recursion the grammar must have no cycles $A \Rightarrow^{+} A$ or null productions $A \rightarrow e$.

For example $A \to Ba \mid b$, $B \to c \mid Ad$ } is indirectly left recursive.

(32) The following algorithm will eliminate these:

1) Arrange the non-terminals in some order A1 A2 ... An

2) FOR i = 1 TO n DO BEGIN
   FOR j = 1 TO i-1 DO

   Replace each production of the form $Ai \to Aj\,y$
   by the productions $Ai \to d1\,y \mid d2\,y \mid \ldots \mid dk\,y$.
   where $Aj \to d1 \mid d2 \mid \ldots \mid dk$ are the current productions for Aj

   Eliminate any direct left recursion amongst the Ai productions

   END.

For example, the grammar $A \to Ba \mid b$.
$B \to Bc \mid Ad \mid e$.

1 - order them A B
2 - check for left recursion in A - none
We now must replace for $B \to Ad$
with $B \to Bc \mid Bad \mid bd \mid e$
Eliminate the direct left recursion in B
$B \to bd\,B1$.
$B1 \to c\,B1 \mid ad\,B1 \mid e$.

Giving the grammar $A \to Ba \mid b$.
$B \to bd\,B1$.
$B1 \to c\,B1 \mid ad\,B1 \mid e$ with no left rec.

(33) # LL(k) GRAMMARS

A top-down parser which can make a deterministic decision about which of several alternative productions with a common LHS to choose when given k lookahead tokens is an LL(k) parser.

LL(k) grammars are a proper subset of context free grammars. They are the largest class that permit deterministic left to right top-down recognition with a look ahead of k symbols. A grammar is LL(k) if the correct production can be deduced from the partially constructed parse tree and the next k tokens in the input string. We would in particular like to have an LL(1) grammar.

To ensure that a grammar $G = (T, N, P, S)$ is LL(1) we must compute the FIRST and FOLLOW sets for all $x \in N$. FIRST(x) is the set of terminals that begin strings derived x. FOLLOW(x) is the set of terminals a that can appear immediately to the right of x in some sentential form. If x can be the rightmost symbol in some sentential form then e is in FOLLOW(x), and similarly if e can be derived from x then e is in FIRST(x).

Given a production $A \to B1 \mid B2 \mid \ldots \mid Bn$. where each B_i can be either terminal or nonterminal; for a grammar to be LL(1), $\bigcap_{i=1}^{n} \text{FIRST}(B_i) = \phi$.

(34) We compute FIRST(x) as

1) If $x \in T$ then $\text{FIRST}(x) = \{x\}$
2) If $x \in N$ and $x \to a\alpha$ is a production, $a \in T$, any α then $a \in \text{FIRST}(x)$.

We compute FOLLOW(x), $x \in N$ as

1) e is in FOLLOW(S)
2) For a production $x \to BCD$ everything in FIRST(D) except e is in FOLLOW(c)
3) For a production $x \to BC$ or $x \to BCD$ where FIRST(D) contains e then everything in FOLLOW(x) is in FOLLOW(c)

Example: The grammar Statement → Expr.
Expr → Term Rest-expr.
Rest-expr → '+' Term Rest-expr | e.
Term → Factor Rest-term.
Rest-term → '*' Factor Rest-term | e.
Factor → '(' Expr ')' | a.

FIRST(Statement) = FIRST(Expr) = FIRST(Term) = FIRST(Factor)
= { (, a }
FIRST(Rest-expr) = { +, e }
FIRST(Rest-term) = { *, e }

FOLLOW(Statement) = { e,) } from rule 1
FOLLOW(Expr) = FOLLOW(Rest-expr) = { e,) }
) from rule 2 applied to Factor →
e from rule 3 applied to Statement →

FOLLOW(Term) = FOLLOW(Rest-term) = { +,), e }
+ from rule 2 applied to Rest-expr →
), e from rule 3 applied to Expr →

FOLLOW(Factor) = { *, +,), e }
* from rule 2 applied to Rest-term →
+,), e from rule 3 applied to Term →

(35) We add an additional symbol into our language, $. This ends all input strings and corresponds to eof. It completes the look-ahead. Adding this to the production:

Statement → Expr $

$ replaces e in all the FOLLOW sets with this altered production.

DEFN: A grammar is LL(1) iff every $x \in N$, including $, has FIRST(x) ∩ FOLLOW(x) = ϕ.

PRODUCING A RECURSIVE DESCENT PARSER

In such a parser every syntactic construct has a procedure to parse and generate code for that construct. This eliminates completely the need for an explicit syntax tree.

Given an LL(1) grammar we can produce a recursive descent parser as follows:

(36) 1) A production $A \rightarrow B1\ B2 \ldots Bn$ is translated to the compound statement:

```
begin
    B1; B2; ... ; Bn
end
```

2) A production $A \rightarrow B1 \mid B2 \mid \ldots \mid Bn$ is translated to

```
case symbol of
    L1 : B1;
    L2 : B2;
    ⋮
    Ln : Bn;
end
```

where L_i is FIRST(B_i)

3) A repetition of the form $A \rightarrow \{B\}$ is translated to:
while symbol in FIRST(B) do B;

4) A repetition of the form $A \rightarrow [B]$ is translated to
if symbol in FIRST(B) then B;

5) A non-terminal of the form $A \rightarrow B$ is translated as a call of procedure B.

6) A terminal of the form $A \rightarrow t$ is translated as:
IF symbol = t then gettoken else error;

Applying these to our sample grammar we get:

```
program parse (input, output);
 var ch: char;

 procedure statement;
   begin
     expression;
     if not (eof) then error1;
   end;

 procedure expression;
   begin
     term;
     rest_of_expression;
   end;

 procedure rest_of_expression;
   begin
```

```
        if ch = '+' then begin
            read (ch);
            term;
            rest_of_expression;
        end;
    end;

    procedure term;
        begin
            factor;
            rest_of_term;
        end;

    procedure rest_of_term;
        begin
            if ch = '*' then begin
                read (ch);
                factor;
                rest_of_term;
            end;
    end;

    procedure factor;
        begin
            if ch = '(' then begin
                read (ch);
                expression;
                if ch <> ')' then error2; else read (ch)
            end
        else if ch = 'a' then read (ch)
        else error 3;
end
```

```
(* Start of main prog *)
begin
    read (ch),
    statement;
end.
```

(40) ## Error Detection and Correction

Error processing encompasses detection, recovery and reporting of errors. Error correction is not feasible as the compiler can only detect an error, not determine what the programmer intended.

Errors in Lexical Analysis

A lexical error can result from problems in identifying a token or reaching the end of file in the middle of a token. Usually the LA is unaware of most errors

If an unidentifiable token, punctuation error or numeric underflow / overflow occurs, the token should be rejected and we should return to state 50 in the FSA. If an eof is encountered in the middle of a prog the the program should be terminated

Recovery from Syntactic Errors

A practical compiler must output an error message as soon as it discovers an ill-formed string and continue the parsing process in an attempt to find further mistakes.

(41) The design of an error recovery mechanism is language

dependent and cannot be generalised for all context free languages.

In languages where the end of record is a statement delimiter eg FORTRAN the error recovery is simple - skip to the end of the record, guaranteeing that only the statement with the error is skipped and checking commences at the next statement.

In a free format block structured language such as PASCAL there are usually certain features specifically designed to facilitate error recovery. The first concerns the language design - it is mandatory that the language contains certain key words which are highly unlikely to be misused and that may therefore serve to bring the parser back into step. (Eg PASCAL - every structured statement begins with an unmistakable keyword such as BEGIN, IF, WHILE, CONST, etc)

The second concerns the structure of the parser. If a procedure in the parser detects an error it should not merely refuse to continue and report back to its calling procedure, but should itself scan text up to a point where some plausible analysis can be resumed. To do this each procedure knows the set of follow symbols at the place of its current activation, and symbols can be skipped until a follow symbol is found. This can have a disastrous effect in places where a follow symbol is
(42) omitted. To prevent this we augment the set of follow symbols with the reserved words that mark the beginning of a construct (cf previous paragraph). Thus the individual parsing procedures work on stopping symbols rather than follow symbols only.

NB - it is more important to generate good error messages than to attempt full recovery.

(43) # STORAGE MANAGEMENT.

Storage must be allocated to
- object program
- user defined data structures
- variables
- constants
- procedure linkage information
- scratch (expression evaluation and parameter transmission)
- I/O buffers

There are two distinct types - static (kept by compiler) and run time machine structures (created by prog execution).

STATIC REPRESENTATION OF DATA OBJECTS

SYMBOL TABLE DESIGN

Each entry is a pair (name, attributes). The attributes could be:
- the type of identifier (eg data location, procedure, label, file, etc)
- if a data location, the value type, location and its links to any data
- if a procedure, the location, parameter list & types, whether user defined or system defined, accessibility, etc
- if a file, the characteristics (eg record size, seqn/random, etc)

A statement whose principal purpose is to assign attributes to an identifier is called a <u>declaration</u>. An identifier is said to be <u>referenced</u> or used in a statement in which it appears but no attributes are added to its attribute set. In general, a declaration affects the symbol table while a reference results in code generation.

(44) An identifier which appears in two declarations in the same scope is said to be multiply declared while an id for which no declaration exists is said to be undeclared. Whether such things are permitted depends on the language.

The info in the symbol table is not required for syntax analysis but is required for semantic analysis, optimisation and code generation. During semantic analysis we check usage consistency and during code generation we need to know the effect of arithmetic operators and the amount of storage space required.

SYMBOL TABLE ORGANISATION

The basic operations are:

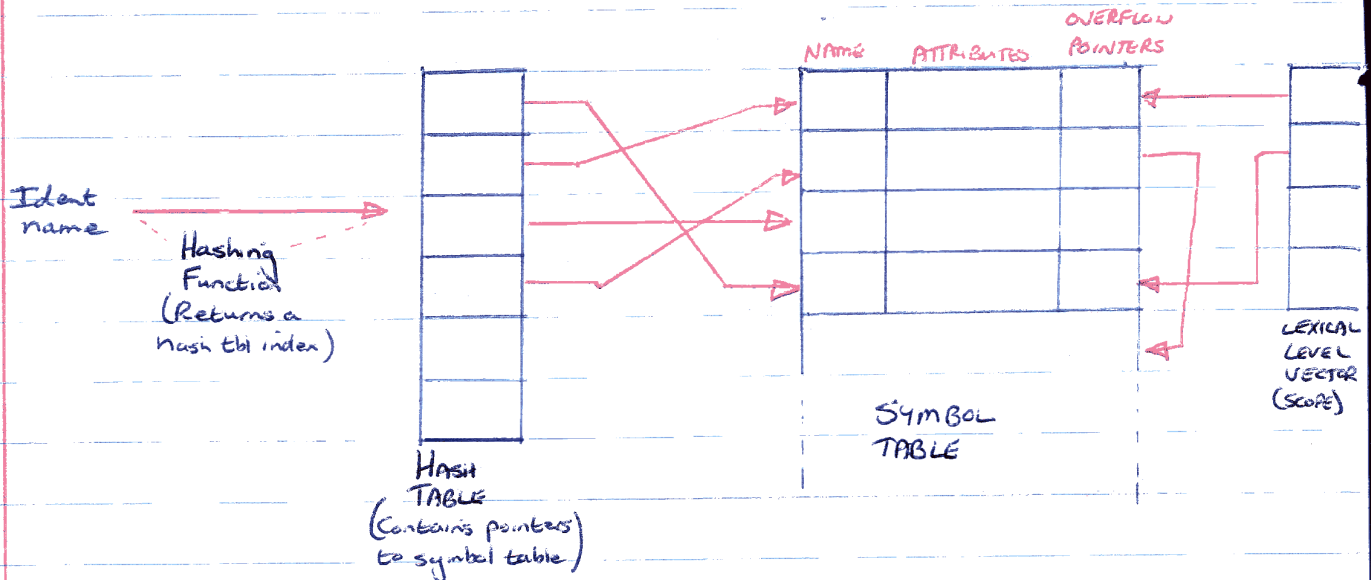
- determine whether a given id is in the table
- add a new id to the table
- access the info associated with an id
- add new info for an id
- delete an id or groups of ids from the table.

The usual methods are linked lists, hash tables and tree structures. In general linked lists are too slow so we will consider the latter two.

HASH TABLE IMPLEMENTATION.

(45) When an identifier is declared an entry is made in the actual symbol table in the next available sequential location. The id name is hashed and the address of the actual symbol table entry is entered into the hash table if no

previous entry exists. If an entry exists a hash collision occurs - we can deal with this by rehashing or by setting up a linked list within the hash table.



46 ## TREE STRUCTURED IMPLEMENTATION.

This normally uses an alphabetically ordered binary tree. This means that by comparing an id name with the name of the node, we can determine whether to search the

47 left or right subtree. If the tree is balanced (ie the length of the subtrees from any node differ by at most 1) then search time is $O(\log n)$. However, this is usually not the case, and, in fact, if the programmer declares variables in alphabetical order, the binary tree will reduce to a linear list. This can be overcome by balancing the tree from time to time, but this is time consuming and usually not necessary. One advantage of a tree structure is that variants can be used (in PASCAL) for different types of identifiers.

(48) ## Representation of Scope in the Symbol Table.

Most languages possess block structured capabilities for limiting the scope of an identifier. This presents two problems

- we must be able to decide which ids in the symbol tbl are accessible
- we must be able to declare ids with the same name in different blocks without getting a doubly declared id error.

The basic concept is to mark the position of the symbol table on entry to a new scope level and expand the symbol table from this point. On exit from this level we can dispose of all entries made after our mark. The start of scope pointers can be held as a vector, the dimension of which will determine the number of nested declarations which can be used.

(50) In hash table representation, upon exit from a scope level, we must sequentially search through both the hash table and the symbol table (examining overflow pointers) to determine which elts can be disposed of, a time consuming process.

In a tree structure we can start a new binary tree for each new scope or lexical level with a vector of pointers indicating the roots of the trees for the various levels. This is considerable simpler as we can simply destroy a tree upon exit from its lexical level.

SYMBOL TABLE INFORMATION.

For every id, we require a name (array of char) and the storage size (eg 1 byte, 1 word, etc) For constants we need a
(51) value; for variables we need a type, a lexical level and a displacement within the lexical level. For procedures and functions we need to know the type (user declared or standard) and if declared, the lexical level, number of parameters, and a pointer to the first parameter in the symbol table.

SYMBOL TABLE ACCESS.

The symbol table is accessed via two routines: enterid which puts a new entry into the table, and searchid which searches through the table one lexical level at a time until (if) it finds the entry for the identifier.

(a) ENTERID

This proc is called with a pointer variable pointing to the
(52) symbol table entry for the id. The record pointed to by this pointer will previously have had all the requisite info entered by the routine handling the ids declaration.
The algorithm is

(a) Start from the current lexical level
(b) Scan through the tree to get to the appropriate point in the tree comparing the name of the id against the name of each node
(c) If the name of the id = name on the table, print out a double declared id message
else place the new entry at the (left or right, whichever is appropriate) tip of the tree and exit

(b) SEARCHID.

This proc is called with the identifier name which on return contains the address of the symbol table entry and the name of the identifier located. The symbol table is searched from the current lexical level down thru successive levels until the entry is found. If no entry is found the null ponte is returned. Type conflict checks can also be performed by the proc.

The algorithm is:

(a) Start from current lexical level.

(b) Scan throug the tree taking left & right branches where appropriate. If id is found, return a ponte to the entry and exit.

(c) If we have searched the entire tree unsuccessfully, go to the tree for the previous lexical level and search it.

(53) (d) If we are at the end of the tree at lexical level 0 set ponte to NIL and exit.

Note: By convention symbol table entries start from lexical level 1 (main program) and go upwards. Level 0 is used for predeclared standard functions (eg READ, WRITE) whose values can be redefined in an inner block but which have a default system value. Thus, prior to parsing the prog. the symbol table must be initialised with all the standard identifiers.

Run-Time Storage Management

Static Storage Allocation.

In a static allocation scheme, it is possible to decide at compile time the address each object will occupy at run time. This requires that the number and size of the possible objects be known at compile time, and that each object may have only one occurrence at a given moment in the execution of the program.

Storage allocation is simple. During pass 1 of the compiler, a symbol table is created in which the name, type, size and address of each object is kept. During code generation (which may be in the same or subsequent passes), the address of each object is thus available for insertion into the object code.

Dynamic Storage Allocation.

Recursive languages preclude any attempt at static storage allocation, as a variable may correspond to several values at a given moment. Also, dynamic array creation adds difficulties. There are two methods of dynamic storage allocation - stack and heap. The stack is for handling recursive procedures, the heap for variable data sizes.

Stack Allocation.

Upon entry to a block or a procedure a new storage allocation is made; upon exit, this space is freed. For

(55) block structured languages, the stack is LIFO, allowing continual utilisation of the available space.

BLOCK LINKAGE.

At any moment, a base register B points to the start of the most recent data block in the stack. B allows reference to be made to all those values which correspond
(56) to local variables. In order to access global (or higher level) variables, the data area corresponding to each block will indicate the start of the preceding block, together with its own block number. This pointer is simply the value of B before creation of the new block. When reference is made to a non-local variable, the compiler produces instructions which descend the chain looking for the block number, and positions a base register on the relevant data area. The non-local value is accessed by displacement from this base register.

The base register points at the word which contains its preceding value. At the end of a block, the stack is returned to its former state by setting B back to this former value. The values declared within the block that has just been left are lost, and the space can be re-used.

However, problems can arise when mixing static and
(57) dynamic blocks, as indicated in the following example.

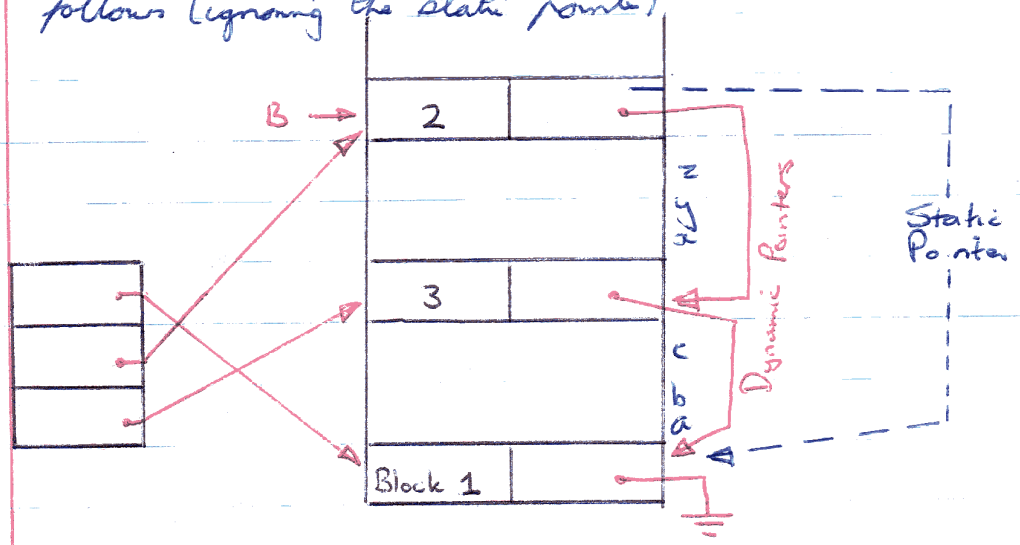
```
procedure A;
var a,b,c : integer;

    procedure Q;
    begin
        a:=b+c;
    end

    procedure C;
    var x, y, z: integer;
    begin
        Q
    end;

begin
    C
end.
```

When the program executes the procedure call, the stack is as follows (ignoring the static pointer)



Within Q, no reference can be made to x, y or z, and no purpose is served by by examining their block when looking for non-local variables. A second pointer should be included in the data area of Q to indicate its statically containing block.

The static pointer is used in searching for non-locals, the dynamic pointer being used to reset the stack at exit from the procedure.

(58) ## DISPLAYS

References to non-local variables can be inefficient with the above method if nesting is deep. One way to avoid this is to have a display table in which are kept pointers to the currently active data blocks corresponding to each block of the program. References to non-local variables are made by displacement from the value of the relevant display.

Extra information needs to be kept in the stack to facilitate the resetting of the display at block or procedure exit.

(59) An improvement to the above scheme is to create a new display at each block or procedure entry, and keep it in the stack. This time the table can contain just those pointers required by the block. The values of the pointers can be deduced at block entry by following the static chain, and this is thus followed only once per block instead of once per non-local reference

(61) # SEMANTIC ANALYSIS.

This phase must deduce the attributes of the various components of a structure, ensure that they are compatible, and then select the proper evaluation procedure from those available.

The tasks of semantic analysis are:

1) Name analysis
2) Finding the definition valid at each use of an identifier
3) Operator identification and type checking determine the operand types and verify that they are allowable for a given operator.

In addition to accessing the type of each we must hold the type of each intermediate value, which will always be the current type. For eg, the expression A + B * C, we will access B and its type, and C and its type, checking that B and C are compatible. We then perform the multiplication, holding the resulting intermediate type. A is then accessed and its type compared with the
(62) intermediate type, and so on. The type checking is normally performed by a boolean function taking as argument two types.

Semantic checking is necessary owing to the inadequacy of a context free grammar to fully define the language. Very few semantic errors can be found at compile time (eg subscript errors, overflows, etc) In general, only type incompatible errors can be found & reported during compilation.

Recovery from semantic errors depends largely on the implementors preference. When semantic errors are detected, code generation should be stopped.

(63) # Code Generation.

The translator must access the symbol table which maps source prog names → run-time objects. This mapping is provided by the object description phase where declarative information is processed, each identifier being associated with a description of a run time object.

We will examine a tree walking translator. This consists of a number of mutually recursive procedures, each of which is capable of translating one kind of source program fragment.

The following example procedures show this translation. Each procedure translates a phrase by generating an instruction or a sequence of instructions which links together the code fragments that are the translation of its sub-phrases. For indivisible phrases (leaf nodes) it is necessary to know how to translate the entire phrase.

Conditional Statement

- Call procedure which generates code to load value of expression into register x.
- Invent a label #Ly
- Generate JUMP FALSE x, #Ly
- Call procedure which generates code to carry out conditional statement part (then ...)
- Generate #Lf:
- Continue (or do else part)

64 Boolean Or (Result → Register x)

- Call proc generating code to load left operand into Rx
- Call proc " " " " right " " Rx+1
- Generate OR x, x+1

Relation '=' (Result → Reg x)

- Call proc generating code to load value of left operand into register x+1
- Call proc generating code " " " " right " " into register x+2
- Generate LOAD x, TRUE
  SKIPEQ x+1, x+2
  LOAD x, FALSE

ARITHMETIC '+' (Result → Reg x)

- Exactly as for Boolean OR, except that the final instruction is ADD x, x+1

ARITHMETIC '*' (Result → Reg x)

- Exactly as for Boolean OR, except the final instruction is MULT x, x+1

Once the translator is producing working code it is possible to look to improving or optimising the code. Very often this is machine dependent.

(66) Translation Error Handler.

If some source fragment defies translation, then an error has occurred and must be reported. The most obvious errors which occur at this stage are undeclared identifiers, type conflicts, etc. Checking for these errors does not slow down translation since types of operands must be checked anyway to select the appropriate code fragment. No error correction during translation should ever be attempted.

(68) Predictive Parsing of LL(1) Grammars

A predictive parser is an efficient way of parsing an LL(1) grammar. It uses a parse table to represent the grammar, and a source-language-independent module of the compiler to proceed through the table according to the input string. In addition, it has a stack used to hold the equivalent of the return addresses each time it enters a new production of some non-terminal.

The parse table is a 2D array with rows corresponding to non-terminals and columns to terminals. The stack contains a sequence of grammar symbols. The end of string and empty stack are both represented by $.

The parser control program has the following actions.

i) If the stack is empty and the current input symbol being analysed is $ then halt and announce successful completion of parsing.

2) If the top item on the stack is a terminal then the current input symbol must be the same terminal else an error has occurred. If the two symbols match, advance to the next symbol of the input string, and pop the stack.

3) If the top item on the stack is a non-terminal then examine PARSE [stack symbol, input symbol]. This will either be a production or a blank entry. Say the entry is the production STACKSYMB → UVW, then the stack symbol is REPLACED by UVW, with U on top. As output, the grammar does the semantic action associated with this production, eg add an entry to the symbol table or produce code.
If the entry in the parse table is blank then the parser takes appropriate error recovery action.

(70) CONSTRUCTION OF PARSE TABLES.

The following algorithm can be used to construct a predictive parsing table for the grammar G.
Suppose $A \to \alpha$ is a production with the terminal a in FIRST(α). Then, whenever the parser has A on top of the stack with a the current input symbol, the parser will expand A by α. The only complication occurs when $\alpha \Rightarrow^* \epsilon$. In this case, we should also expand A by α if the current input symbol is in FOLLOW(A), or if the \$ on the input has been reached and ϵ is in FOLLOW(A).

The algorithm is

1) For each production $A \to \alpha$, do steps 2 & 3

2) For each terminal $a \in FIRST(\alpha)$, add Parse$[A, a] = \{A \to \alpha\}$

3) If $e \in FIRST(\alpha)$, then for each terminal $b \in FOLLOW(A)$,
add Parse$[A, b] = \{A \to \alpha\}$
If $e \in FIRST(\alpha)$ and $e \in FOLLOW(A)$, add Parse$[A, \$] = \{A \to \alpha\}$

(71) Example: the grammar STATEMENT → IF condition THEN statement
REMAINDER | a.
REMAINDER → ELSE statement | e.
Condition → b.

This gives the sets:

FIRST (STATEMENT) = {IF, a}
FIRST (REMAINDER) = {ELSE, e}
FIRST (CONDITION) = {b}

FOLLOW (STATEMENT) = {e, ELSE}
FOLLOW (REMAINDER) = {e} ?
FOLLOW (CONDITION) = {THEN}

This gives the parse table (note that there's a multiple entry – an LL(1) grammar will have no multiple entries in its parse table):

	a	b	IF	THEN	ELSE	$
STATEMENT	STATEMENT → a		STATEMENT → IF...			
REMAINDER					REMAINDER → e* / REMAINDER → ELSE	REMAINDER → e
CONDITION		CONDITION → b				

* this follows from rule 3 (all others rule 2) and is a result of an ambiguity in the grammar.