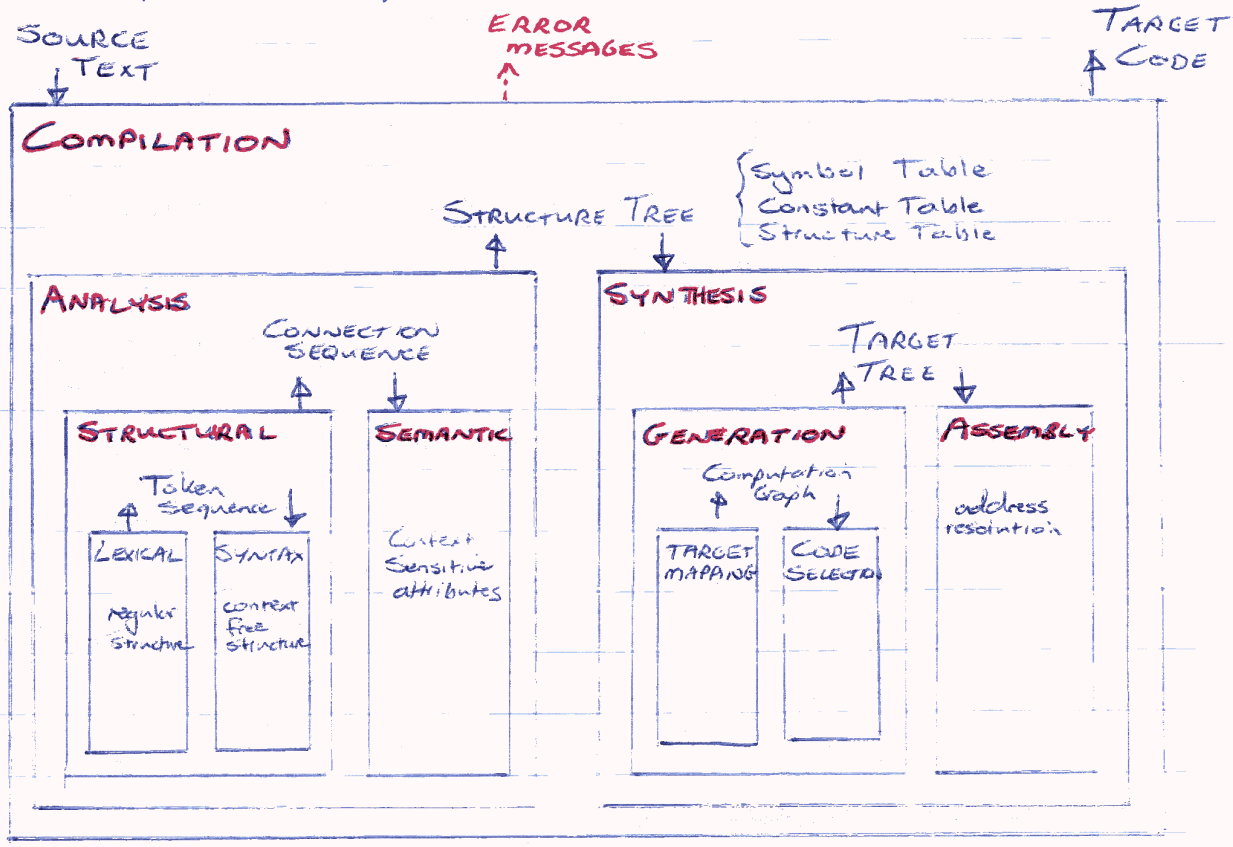
Formal Semantics I

Lectured : T.S. McDermott 24 Feb 1986 – 1 April 1986

Analysis ↑ — Synthesis ↓

(1) General Structure of a Modularised Compiler

An abstract structure of a modularised compiler is shown below - abstract as could be one-pass (parallel-) or two-pass (sequential module execution).



MODULE :- an action without a state (ie procedures have no internal state, at least in their mathematical defns ?!) local variables protected from above routines

(2) Phrase Structure Grammars & Languages

V = Vocabulary (alphabet) set of symbols

L = Language subset of V^* (strings over V)

G = Grammar generator of language (grammar finite but strings produced not necessarily)

$L \equiv \{ X : S \overset{+}{\Rightarrow} X \}$ S a start symbol

$X \in T^*$ terminal strings

V - vocab = N ∪ T
P - productions
T - terminals
Z - single start symbol
N - non-terminals

(V, P) form a rewriting system
(T, N, P, Z) a phrase structure (Chomsky) grammar for $\{X \in T^* : Z \overset{+}{\Rightarrow} X\}$ language.

Chomsky uses only sets which are grammarable or enumerable.

most restrictive Type 3 ⊂ Type 2 ⊂ Type 1 ⊂ Type 0 most general

Acceptors for Phrase Structure Grammars (Automata)

Given a string, is it in the language defined by the grammar? Much formal work has been done relating classes of grammars to classes of automata. This relationship is intimate, however for a grammar to be useful to a compiler writer, it must be computationally tractable as well as reasonably easy to generate the parse tree for any input string. Type 3 & 2 languages can be thus parsed, Type 1 & 0 present fantastic problems for automatic parsers.

Type 3 (Finite State) Grammars. (also Regular Grammars)

Rules of two forms only: $N' \to T' \mid N''T''$
Exactly one parse tree.

Type 2 (Context Free) Grammars

Rules of form $N \to S$ where $S \in T^*$. We can generate semantically meaningful trees (eg operator precedences)

Type 1 (Context Sensitive) Grammars

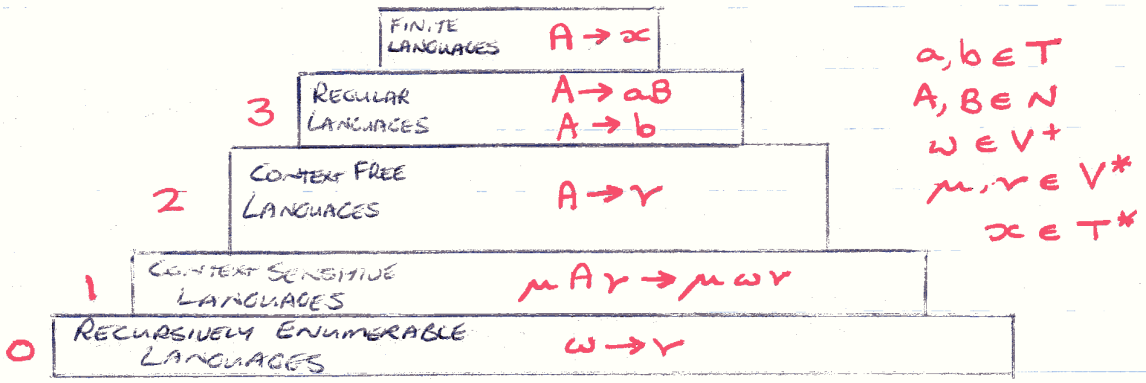
$xAy \Rightarrow xSy \quad A \in N,\ S \in V^+,\ x, y \in V^*$

As $S \in V^+$ RHS always longer or same length as LHS.
This property means that type 1 grammars are decidable ie given a string we can decide if it is in the language

Type 0 (Phrase Structure) Grammars (also Recursively Enumerable)

$w \rightarrow \mu \quad$ where $w \in V^+ ;\ \mu \in V^*$

Summary of the Classification of Phrase-Structure G's.



Phrase Structure & Meaning

A phrase is a substring which derives from a single non-terminal. We interpret a phrase as a semantic unit, ie we are not satisfied with any grammar, we choose one upon which we can base a semantic analysis of our sentences. For this reason we use context-free grammars

We lose this semantic content in context sensitive languages. For example, $a^n b^n a^n$ (which cannot be produced by context free) can be produced with a context sensitive grammar

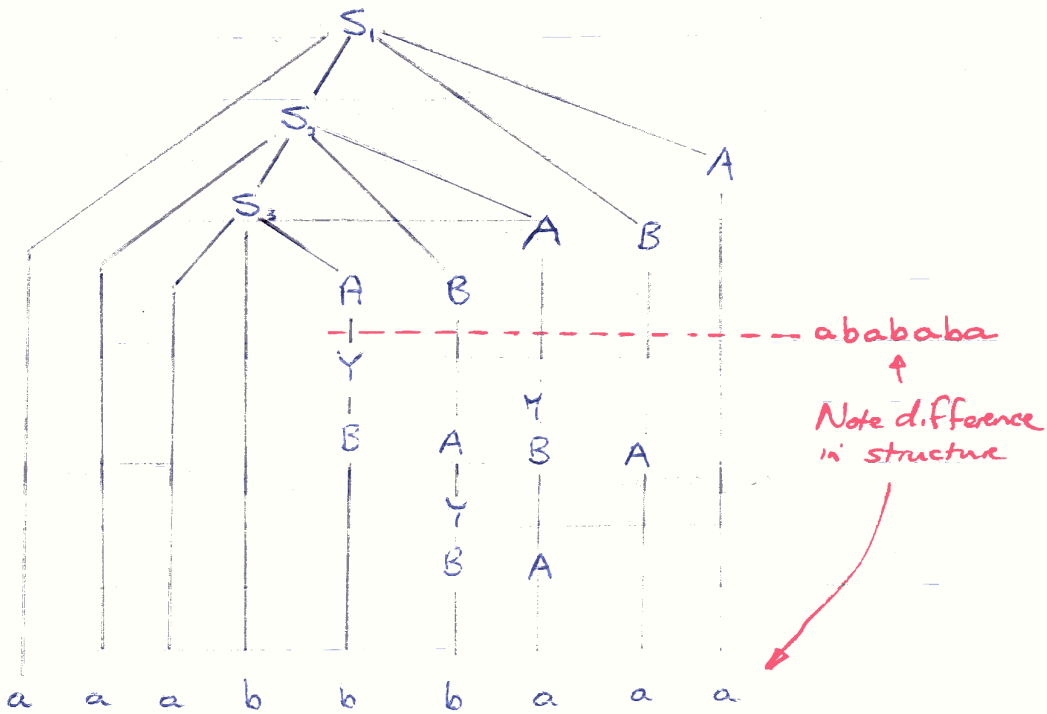
$S \rightarrow aSBA \mid abA$

$bB \rightarrow bb$

$A \rightarrow a$

$AB \rightarrow YB \quad YB \rightarrow YA \quad YA \rightarrow BA \quad [= AB \rightarrow BA]$

eg



Note that the non-terminals no longer represent some 'abstracted' characterisation of the structure of the string they derive. $S_1 \overset{*}{\Rightarrow} aaabbbaaa \quad S_2 \overset{*}{\Rightarrow} aabba \quad S_3 \overset{*}{\Rightarrow} abb$

NB Chomsky considered CS to be phrase-structure but for our purposes we consider CF to be, not CS. Chomsky does not deal with semantics but rather with strings which can be analysed mathematically.

WHY CHOMSKY GRAMMARS?

1. over against BNF unifies the description of all types of grammarable languages (eg unified notion of production)
2. obscures difference of generative analytic syntax
3. obscures difference of constituency & dependency syntax
4. relationship to automata and formal systems

startsymbol ≡ axioms
productions ≡ rules of inference
derivation ≡ proof
terminal strings ≡ theorems

Points 2 & 3 come from the 2 disjoint alphabets, 1 startsymbol and 1-place rules.

eg

<subj> <pred> <obj>

They made him do it

<subj> <pred> <obj>

(3) PARSING

3.1

We can parse various grammars using automata

③ regular grammars ≡ finite automata
① context sensitive ≡ push-down automata
② context free ≡ limit bounded automata ?
⓪ recursively enumerable ≡ Turing Machines

Parsing Regular Grammars with Finite State Automata

RG = (T, N, P, Z)

FSA = (T, N, R, Z, F) R = productions F = final state, F ∈ N.

(T ∪ N, R) form a rewriting system.

If A → aB ∈ P, then Aa → B ∈ R.

(eg R = { Ch → q|F. C. → I. F → I. FE → S Ih → q|X. }

P = { C → n|nF|.I. F. → I|ES. I → n|nX X → EC.

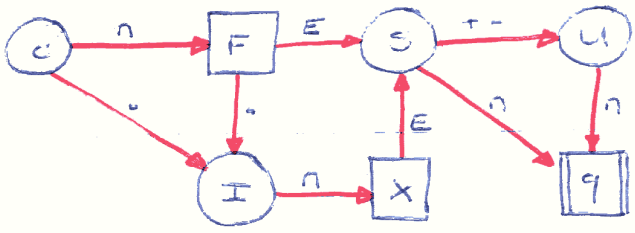
S → +U|-U|n. U → n }

Z = C F = {q}

a regular grammar for real numbers.

All non-terminals are states and q is the empty non-terminal.

This is a non-deterministic FSA but can be changed into a deterministic FSA (resulting in a state being a set of non-terminals, not just one). Each non-terminal represents the rest of the control string (or at least what it is expected to be).



NDFSA for recognising real numbers

Self-Embedding

Nesting (self-embedding) cannot be done by regular grammars, but requires context free grammars, which can be parsed by pushdown automata using a stack.

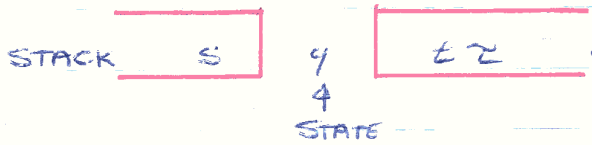
PDA = (T, Q, S, R, q_0, S_0, F) S = set of states of stack

q_0 : start symbol Q = set of states of control string

S_0 = start stack

(T ∪ Q ∪ S, R) form a rewriting system.

Each member of R has the form $sqtz \rightarrow s'q'z$



Note that:

FSA has a finite number of states

PDA has a finite number of stack states, but an infinite number of parser states

DPDA has an infinite number of parser states, but these are partitioned into a finite number of equivalence classes. The language of the stack is a finite state automaton

DETERMINISTIC PARSERS

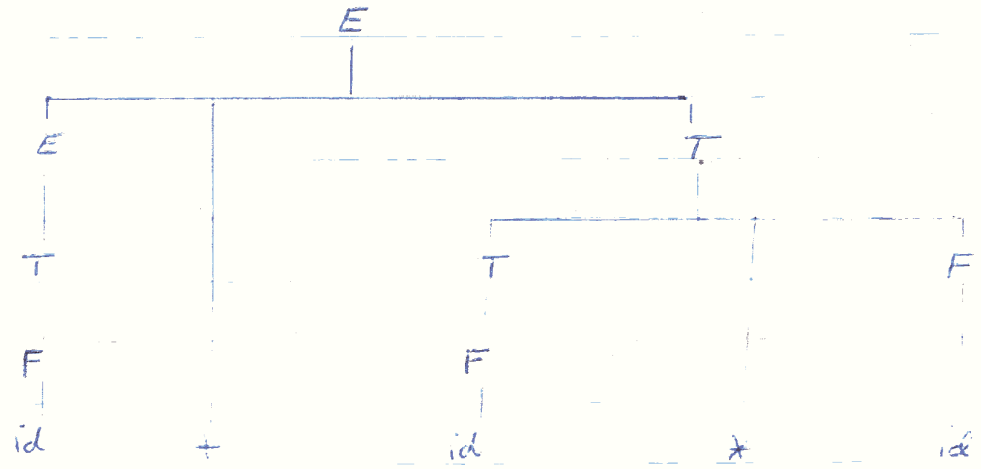
- TOP DOWN – begin with start symbol and try to derive the null string – produces a <u>left derivation</u>

- BOTTOM UP – begin with an empty stack and try to produce the start symbol, <u>shifting</u> symbols onto the stack and trying to <u>reduce</u> the contents of the stack (see eg overleaf) produces <u>right derivation</u>

WHY DETERMINISTIC?

To parse without backtracking. For a regular grammar we can always find a deterministic grammar. As PDA's can have an infinite number of parser states we cannot always construct deterministic grammars. All computer languages are designed to be deterministically parsable.

$E \to T \mid E+T$ $\quad T \to F \mid T*F$ $\quad F \to (E) \mid id$



TOPDOWN

Stack	Control
E	$i+i*i$
$T+E$	$i+i*i$
$T+T$	$i+i*i$
$T+F$	$i+i*i$
$T+i$	$i+i*i$
$F*T$	$i*i$
$F*F$	$i*i$
$F*i$	$i*i$
i	i
Λ	Λ

DERIVATION

Leftmost	Rightmost
E	E
$E+T$	$E+T$
$T+T$	$E+T*F$
$F+T$	$E+T*i$
$i+T$	$E+F*i$
$i+T*F$	$E+i*i$
$i+F*F$	$T+i*i$
$i+i*F$	$F+i*i$
$i+i*i$	$i+i*i$

BOTTOM UP

Stack	Control
E	Λ
$E+T$	Λ
$E+T*F$	Λ
$E+T*i$	Λ
$E+F$	$*i$
$E+i$	$*i$
T	$+i*i$
F	$+i*i$
i	$+i*i$
Λ	$i+i*i$

LL & LR

The letters define the directions of scan control and derivation stack.

3.2 LR PARSING

In every LR-parser there is a FSA controlling the stack. E is called the dominant non-terminal of the grammar, G the preliminary start symbol, and cannot appear on the right of a production. The follow set is the set of terminals which can occur after a state containing the non-terminal.

GRAMMAR

$G \to E$ $\quad T \to F \mid T*F$

$E \to E+T \mid T$ $\quad F \to (E) \mid id$

(refer bottom up demo opposite)

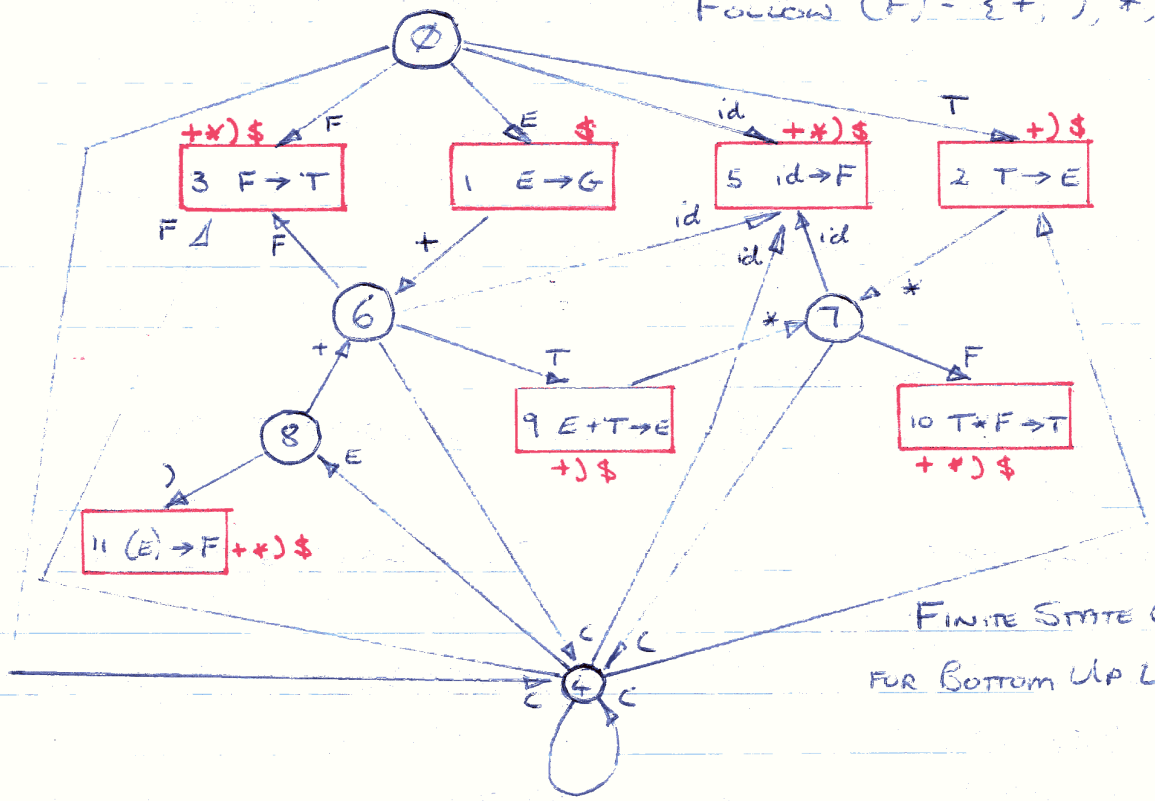
FOLLOW SETS

Follow(G) = {$}

Follow(E) = {+,), $}

Follow(T) = {+,), *, $}

Follow(F) = {+,), *, $}



FINITE STATE CONTROL FOR BOTTOM UP LR PARSER.

Note: State 5 is a reduce state only, reducing id to F then going back to state 0. At each reduce state, if a shift can be done it is else back to state 0. The red letters show the follow sets \ the shift symbols (i.e. those symbols which force a return to state 0)

○ shift state ▭ reduce state

Constructing the FSA

Defns

Item - an item contains a non-terminal followed by a stack and control in the form.
<non-terminal> → <stack> . <control>

item.set - a set of items associated with a state (continued later)

LR Parser Implementation

LR Parsers generally consist of a driver and table (usually machine generated). We will consider 3 types: SLR (simple LR)
LR (canonical LR)
LALR (look-ahead LR)

LR Parsers

An LR parser has an input, a stack, and a parsing table. The input is read from left to right, one symbol at a time. The stack contains a string of the form $S_0 X_1 S_1 X_2 \ldots X_m S_m$, where S_m is on top. Each X_i is a grammar symbol and each S_i a state. State summarizes the information in the stack and guide the shift/reduce action. The parsing table consists of two parts, a parsing function action and a control function goto.

The driver consults entry action $[S_m, a_i]$ where a_i is the current input symbol. This entry can have one of four values.

- shift s
- reduce $A \to \beta$
- accept (parsing complete)
- error

goto (<state>, <grammar symbol>) produces the new state

All LR parsers behave in this way: only the table construction methods vary. A parsing table for our example grammar is:

Grammar (productions numbered)

(0) $G \to E$ 1,2 $E \to E+T \mid E$ 3,4 $T \to T*F \mid F$
5,6 $F \to (E) \mid id$

State	Action (Shift/Reduce)						Goto (state)		
	id	+	*	(	)	$	E	T	F
0	s4			s4					
1		s6				accept			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

All empty actions are errors

sk shift & goto state k

rk reduce according to production k

NB refer to Wurb & Yoos, pp 176 ff for implement. of sparse table.

A grammar for which we can construct an LR parsing table in which every entry is uniquely defined is an LR grammar. There are context-free grammars which are not LR but these are avoided (eg any ambiguous grammar)*. The driver routine of an LR parser is essentially a finite automaton.
Note that the LR requirement that we be able to recognise the occurrence of the right side of a production having seen what is derived from that right side is far less stringent than the requirement for a predictive parser, namely that we be able to recognise the apparent use of the production seeing only the first symbol it derives.

3.3 CONSTRUCTING THE DFA (LR(0)) (ie CANONICAL LR(0) ITEMSETS)

We show how to construct a DFA from the grammar, which can then be converted into a parsing table. The DFA recognises viable prefixes of the grammar, ie prefixes of right sentential forms that do not contain any symbols to the right of the handle. A viable prefix we so called because it is always possible to add terminal symbols to the right of a viable prefix to obtain a right sentential form. There is therefore no apparent error provided the input to a given point can be reduced to a viable prefix.

We define an LR(0) item of a grammar G to be a production of G with a dot at some position of the right

* NB an ambiguous grammar can be parsed if shifting is given precedence over reducing.

side Eg $A \to XYZ$ generates four items: $A \to .XYZ$, $A \to X.YZ$ $A \to XY.Z$ and $A \to XYZ.$ The production $A \to \varepsilon$ generates only one item, $A \to .$ In an implementation we can represent an item by a pair (<production #><position>)
We group items together into sets which give rise to the states of an LR parser.
One collection of sets of items, which we call the canonical $IR(0)$ collection, provides the basis for constructing SLR (simple LR) parsers. To construct the canonical $LR(0)$ collection for a grammar, we need to define an augmented grammar and two functions CLOSURE and GOTO.
If G is a grammar with a start symbol S, then G' the augmented grammar for G, is G with a new start symbol S' and production $S' \to S$. This is so that the parser knows that when it reduces S to S' it is going into the accept state.

Closure of the Itemset

If I is a set of items for a grammar G, then the set of items CLOSURE (I) is constructed from I by:
- every item in I is in CLOSURE (I)
- if $A \to \alpha.B\beta \in$ CLOSURE (I) and $B \to \gamma$ is a production then $B \to .\gamma$ is in CLOSURE (I)

Intuitively $A \to \alpha.B\beta \in$ CLOSURE (I) indicates that at some point in the parsing process we need expect to see a string derivable from $B\beta$ as input. If $B \to \gamma$ is a production, we could also expect to see a string derivable from γ at this point.

GOTO

The second useful function is GOTO (I, X) where I is an itemset and X a grammar symbol. GOTO (I, X) is defined to be the closure of the set of all items $A \to \alpha X \cdot \beta$ such that $A \to \alpha \cdot X\beta$ is in I.

Intuitively, if I is the set of items that are valid for some viable prefix γ, then GOTO (I, X) is the set of items that are valid for the viable prefix γX.

Example: Consider the augmented grammar:

$E' \to E$ $\quad$ $T \to T * F \mid F$

$E \to E + T \mid T$ $\quad$ $F \to id \mid (E)$

If I is the set of one item $\{[E' \to \cdot E]\}$ then closure (I) contains the items:

$E' \to \cdot E$ $\quad$ $E \to \cdot E + T$ $\quad$ $E \to \cdot T$

$T \to \cdot T * F$ $\quad$ $T \to \cdot F$ $\quad$ $F \to \cdot (E)$ $\quad$ $F \to \cdot id$

If I is the set of items $\{[E' \to \cdot E], [E \to E \cdot + T]\}$ then GOTO $(I, +)$ is the closure of $\{[E \to E + \cdot T]\}$.

We can now give the algorithm to construct C, the canonical collection of sets of LR(0) items for an augmented grammar G' (see opposite).

We shall now construct an LR(0) parser for our grammar. We begin with $E' \to \cdot E$ and take the closure of this - this gives the itemset for state 0, which is

$I_0 = \{[E' \to \cdot E], [E \to \cdot E + T], [E \to \cdot T], [E \to \cdot T * F]$

$[T \to \cdot F], [F \to \cdot (E)], [F \to \cdot id]\}$

```
procedure ITEMS (G');
begin
    C := {closure ({S' → .S})};
    repeat
        for each itemset I and grammar symbol X
            if GOTO (I, X) is non-empty
                then C := C ∪ GOTO (I, X)
    until all itemsets and grammar symbols have been done
end
```

Algorithm to construct the canonical collection of LR(0) itemsets.
(above - general alg ; below FSA alg)

begin with state 0 } (ie C := {closure ({S' → .S})},)
close item set
work out edges (ie x's) and produce their item sets
for all new item sets, we have new states
Repeat for each state until there are no more states

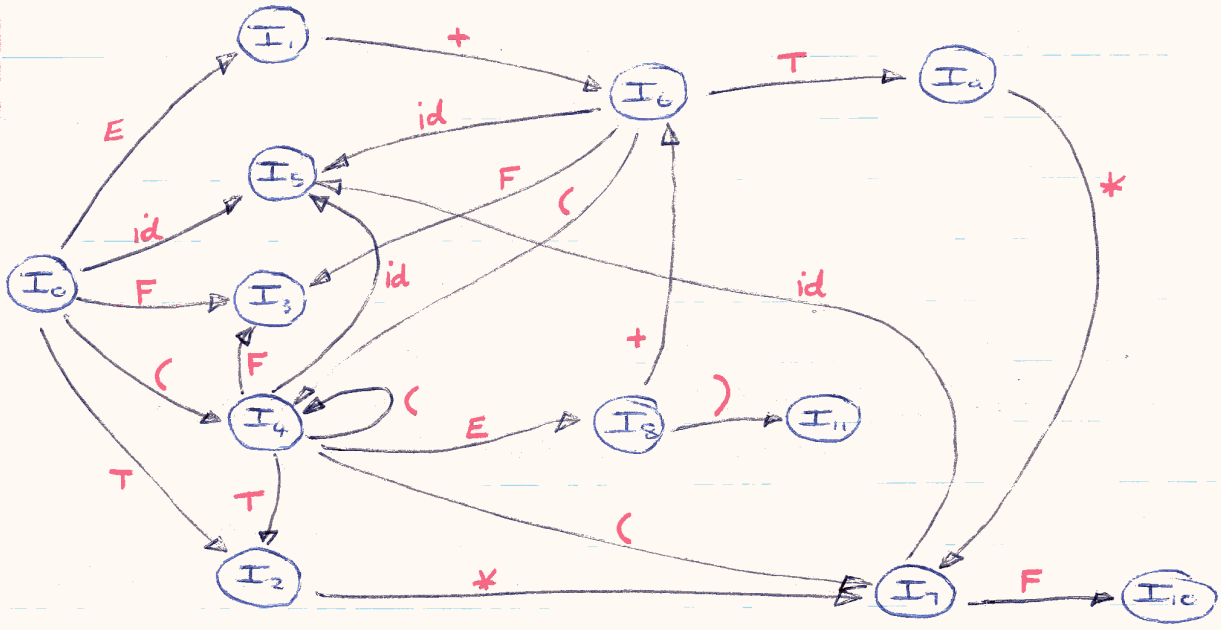
State 0 has edges E, T, F, (and id. Consider edge E. Then GOTO (I_0, E) is closure $\{[E' \to E.], [E \to E. + T]\}$
Let this be state 1. Then $I_1 = \{[E' \to E.], [E \to E. + T]\}$
In a similar fashion we obtain:
$I_2 = \{[E \to T.], [T \to T. * F]\}$ where edge 2 is T
$I_3 = \{[T \to F.]\}$ where edge 3 is F
$I_4 = \{\ldots\}$ where edge 4 is (
$I_5 = \{[F \to id.]\}$ where edge 5 is id.

We then work out I_6 from GOTO $[I_1, +]$, I_7 from GOTO $[I_7, *]$, I_8 from GOTO $[I_4, E]$, GOTO $[I_4, T]$, GOTO $[I_4, F]$, GOTO $[I_4, (]$ and GOTO $[I_4, id]$. I_9 comes from GOTO $[I_6, n]$ where $n \in \{T, F, (, id\}$, I_{10} from I_7 and F, (, id ; and I_{11} from I_8 with) + and *.

This gives us the pushdown parser :

edge	state	item	closure
	0	$\{[E' \to .E]\}$	$\{[E' \to .E], [E \to .E+T], [E \to .T], [T \to .T*F], [T \to .F], [F \to .(E)], [F \to .id]\}$
0E	1	$\{[E' \to E.], [E \to E.+T]\}$	$\{[E' \to E.], [E \to E.+T]\}$
0T	2	$\{[E \to T.], [T \to T.*F]\}$	$\{[E \to T.], [T \to T.*F]\}$
0F	3	$\{[T \to F.]\}$	$\{[T \to F.]\}$
0(	4	$\{[F \to (.E)]\}$	$\{[F \to (.E)], [E \to .E+T], [E \to .T], [T \to .T*F], [T \to .F], [F \to .(E)], [F \to .id]\}$
0id	5	$\{[F \to id.]\}$	$\{[F \to id.]\}$
1+	6	$\{[E \to E+.T]\}$	$\{[E \to E+.T], [T \to .T*F], [T \to .F], [F \to .(E)], [F \to .id]\}$
2*	7	$\{[T \to T*.F]\}$	$\{[T \to T*.F], [F \to .(E)], [F \to .id]\}$
4E	8	$\{[F \to (E.)], [E \to E.+T]\}$	$\{[F \to (E.)], [E \to E.+T]\}$
4T		= 2	
4F		= 3	
4(		= 4	
4id		= 5	
6T	9	$\{[E \to E+T.], [T \to T.*F]\}$	$\{[E \to E+T.], [T \to T.*F]\}$
6F		= 3	
6(		= 4	
6id		= 5	
7F	10	$\{[T \to T*F.]\}$	$\{[T \to T*F.]\}$
7(		= 4	
7id		= 5	
8)	11	$\{[F \to (E).]\}$	$\{[F \to (E).]\}$
8+		= 6	
9*		= 7	

which becomes the DPDA shown opposite. Note that, for example, state 1 is ambiguous as one item specifies a shift and the other a reduce.



DSFA for Grammar G'.

3.4 Constructing SLR Parsing Tables

We want to construct action and goto tables from our DPDA. The parsing actions for state i are determined as follows

- if $[A \to \alpha . a\beta]$ is in I_i and GOTO $(I_i, a) = I_j$ then action $[i, a]$:= "shift j"
- if $[A \to \alpha .]$ is in I_i then for all $a \in$ Follow (A) set action $[i, a]$ = "reduce $A \to \alpha$"
- if $[S' \to S.]$ is in I_i, then set action $[i, \$]$:= "accept".

If any conflicting actions are generated, the grammar is not SLR (1), and the algorithm fails to produce a valid parser. We consult follow set to decide whether to reduce.

The goto transitions for state i are constructed using:

- if GOTO $(I_i, A) = I_j$ then goto $[i, A] = j$

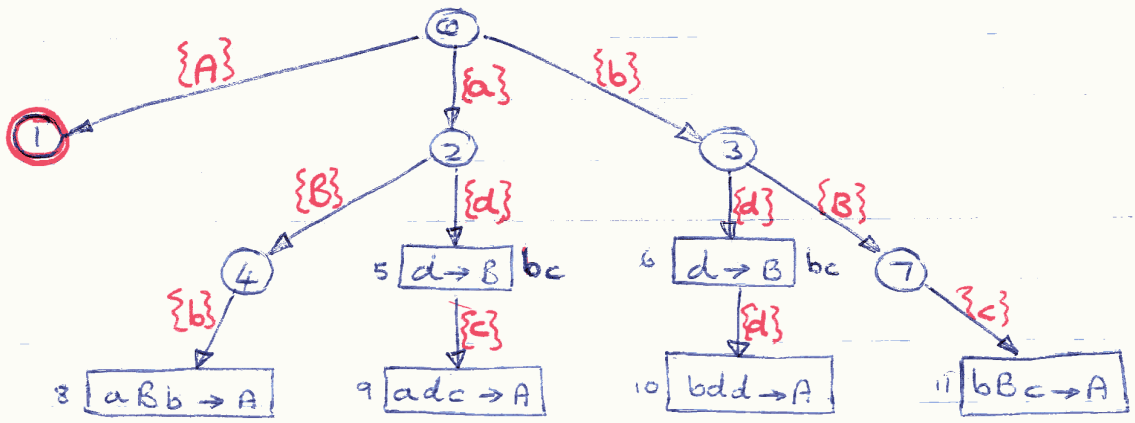
The tables for our example are given on page 11

A problem with SLR is that one can end up with a non-deterministic machine even if the grammar is unambiguous.

Eg: $Z \rightarrow A$
$A \rightarrow aBb \mid adc \mid bBc \mid bdd$
$B \rightarrow d$

Follow(B) = {b, c}

This gives the FSA:



The problem occurs when the transition from state 2 to state 5 is made. The followset of B in this case is not {b, c} but only {b}. If we use {b, c}, we could reduce d to B (state 5) but this prevents us from recognising the correct string in state 9 as we now have aBc rather than adc. This problem is solved in LALR where we consult the follow set for a particular state, eg $Follow_5(B) = \{b\}$ (from $A \rightarrow aBb$) and $Follow_6(B) = \{c\}$ (from $A \rightarrow bBc$)

3.5 Constructing Canonical LR Parsing Tables.

In the SLR method, state i calls for a reduction by $A \to \alpha$ if some set of items I_i contains item $[A \to \alpha \cdot]$ and a is in follow(A). In some situations, however, when state i appears on top of the stack, the viable prefix (cf p12) $\beta\alpha$ on the stack is such that βA cannot be followed by a in a right-sentential form (cf opposite). Thus the reduction would be invalid on input a.

It is possible to carry more information in the state that will allow us to rule out some of these invalid reductions by $A \to \alpha$. By splitting states when necessary, we can arrange to have each state of an LR parser indicate exactly which input symbols can follow a handle α for which there is a possible reduction to A. The extra information is incorporated into the state by redefining items to include a terminal symbol as a second component. The general form of an item becomes $[A \to \alpha \cdot \beta, a]$; we call such an item an <u>LR(1) item</u>. The lookahead has no effect if β is not ε, but in an item of the form $[A \to \alpha \cdot, a]$ we reduce by $A \to \alpha$ only if the next input symbol is a. Thus, we reduce by $A \to \alpha$ <u>only on those input symbols a for which $[A \to \alpha \cdot, a]$ is an LR(1) item in the state on top of the stack.</u>

Formally, we say LR(1) item $[A \to \alpha \cdot \beta, a]$ is <u>valid</u> for viable prefix γ if there is a derivation $S \overset{*}{\underset{rm}{\Rightarrow}} \delta A w \underset{rm}{\Rightarrow} \delta\alpha\beta w$ where $\gamma = \delta\alpha$ and

either a is the first symbol of w, or w is ε and a is \$

The method for constructing the collection of sets of valid LR(1)

items is essentially the same as for canonical LR(0) items (cf 3.3).

To appreciate the definition of the closure operation, consider an item of the form $[A \to \alpha . B\beta, a]$ in the set of items valid for some viable prefix γ. Then there is a rightmost derivation $S \overset{*}{\underset{rm}{\Rightarrow}} \delta A a x \underset{rm}{\Rightarrow} \delta\alpha B\beta a x$ where $\gamma = \delta\alpha$. Suppose $\beta a x$ derives terminal string by. Then for each production of the form $B \to \eta$ for some η, we have derivation $S \overset{*}{\underset{rm}{\Rightarrow}} \gamma B by \underset{rm}{\Rightarrow} \gamma\eta by$. Thus $[B \to . \eta, b]$ is valid for γ. Note that b can be the first terminal derived from β, or a if β derives ε in the derivation $\beta a x \Rightarrow by$, in which case b = a. To summarize both possibilities we say that b can be any terminal in FIRST($\beta a x$), as x cannot contain the first terminal of by, FIRST($\beta a x$) is FIRST(βa). We can now give the algorithms.

```
procedure CLOSURE (I)
begin
    repeat
        for each item [A → α . Bβ, a] in I, each
            production B → γ and each terminal b in
            FIRST (βa) such that [B → . γ, b] is not in I
        do add [B → . γ, b] to I
    until no more items can be added to I
return (I)
end

procedure GOTO (I, X)
begin
    let J be the set of items [A → αX . β, a] such that
            [A → α . Xβ, a] is in I
    return (closure (J))
end
```

```
begin
    C := closure({[S' → .S, $]})
    repeat
        for each set of items I in C and each grammar
            symbol X such that GOTO(I, X) is not empty
        do C := C ∪ GOTO(I, X)
    until no more sets of items can be added to C
end
```

Algorithm to construct LR(1) itemsets.

The algorithm to construct LR(1) parse tables follows:

- construct $C = \{I_0, I_1, \ldots, I_n\}$ the collection of itemsets for G'
- state i of the parser is constructed from I_i. The parsing actions are determined as follows:
  - if $[A \to \alpha . a\beta, b] \in I_i \wedge GOTO(I_i, a) = I_j$ then action $[i, a]$:= "shift j"
  - if $[A \to \alpha ., a] \in I_i$ then action $[i, a]$:= "reduce $A \to \alpha$"
  - if $[S' \to S., \$] \in I_i$, then action $[i, \$]$:= "accept".

If a conflict results from the above rules, the grammar is said not to be LR(1) and the algorithm fails.

- the goto transitions are determined by:
  if $GOTO(I_i, A) = I_j$ then goto $[i, A] = j$

The table formed by this algorithm is called the canonical LR(1) parsing table. An LR parser using this table is a canonical LR parser. If the parsing action function has no multiply defined entries, then the grammar is an LR(1) grammar. Every SLR(1) grammar is an LR(1) grammar, but the canonical parser may have more states than the SLR parser for the same grammar.

3.6 Constructing LALR Parsing Tables.

This is, in practice, the method of choice as the tables it produces are considerably smaller than the canonical LR tables, yet most common PL syntactic constructs can be expressed conveniently by an LALR grammar.

We look for sets of LR(1) items having the same core, that is, set of first components, and we may merge these sets with common cores into one set of items.

eg: $C \to d.$, c/d have core $\{ C \to d. \}$
$C \to d.$, $

Note that, in general, a core is a set of LR(0) items for the grammar at hand, and that an LR(1) grammar may produce more than two sets of items with the same core.

Since the core of GOTO (I, x) depends only on the core of I, the gotos of merged sets can themselves be merged. Suppose we have an LR(1) grammar (ie no parsing conflicts). If we replace all states having the same core with their union, it is possible that there is a conflict, but unlikely because:

Suppose in the union there is a conflict on lookahead a because there is an item $[A \to \alpha., a]$ calling for reduction by $A \to \alpha$, and there is another item $[B \to \beta.a\gamma, b]$ calling for a shift. Then some set of items from which the union was formed has item $[A \to \alpha., a]$, and since the cores of all these states are the same, it must have an item $[B \to \beta.a\gamma, c]$ for some c. But then this state has the same shift-reduce conflict on a, and the grammar was not LR(1) as we assumed. Thus the merging of states with a common core can never produce a SR conflict because shift actions depend only

on the core, not the lookahead.

It is possible, however, that a merger will produce a reduce-reduce conflict, as the following example shows.

Consider the gramma $S' \to S$

$S \to aAd \mid bBd \mid aBe \mid bAe$

$A \to c \qquad B \to c$

generating the four strings acd, ace, bcd and bce. This grammar is LR(1). If we construct the sets of items, we find $\{[A \to c., d], [B \to c., e]\}$ valid for viable prefix ac and $\{[A \to c., e], [B \to c., d]\}$ valid for bc. Neither of these sets has a conflict, and their cores are the same. The union $\{[A \to c., d/e], [B \to c., d/e]\}$ does have a conflict, since reductions by both $A \to c$ and $B \to c$ are called for on inputs d and e.

An easy, but space consuming LALR(1) table construction method follows, which serves primarily as a definition of LALR(1) grammars. Constructing the entire collection of LR(1) sets of items requires too much space & time to be used in practice.

- Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(1) items.
- For each core present amongst the sets of LR(1) items, find all sets having that core, and replace these sets by their union.
- Let $C' = \{J_0, J_1, \ldots, J_m\}$ be the resulting sets of LR(1) items. The parsing actions for state i are constructed as before for CLR.
- The goto table is constructed as follows: if J is the union of one or more sets of LR(1) items, i.e. $J = I_1 \cup I_2 \cup \ldots \cup I_k$, then the cores of $GOTO(I_1, X), GOTO(I_2, X), \ldots, GOTO(I_k, X)$ are the same as $I_1, I_2, \ldots, I_k$ all have the same core. Then $GOTO(J, X)$ is their union.

Eg

Construction of LR(1) parser

Grammar $G \to E$ $\quad T \to T * F \mid F$

$E \to E + T \mid T$ $\quad F \to (E) \mid id$

Note (for LR(1)): An itemset is closed if (assuming there is a production $B \to x$ it contains $[B \to .x, t]$ whenever it contains either

- $[A \to \alpha . B t \gamma]$
- or $[A \to \alpha . B, t]$
- or $[A \to \lambda . B C \gamma, u]$, and $t \in FIRST(C)$

In the next state $\bullet$ moves past B without change of right context. The first state corresponds to the closure of $[G \to .E, \$]$

An itemset is <u>inadequate</u> if it contains an item $[A \to B., t]$ and either another item $[C \to D., t]$ (reduce/reduce conflict) or an item $[C \to D. tE, \tau]$ (shift/reduce conflict) eg state 2 is inadequate on the LR(0)

edge	state	item	closure
	0	$\{[G \to .E, \$]\}$	$\{[G \to .E, \$], [E \to .E+T, \$], [E \to .T, \$], [E \to .E+T, +]$ (etc.)
		or write	$\{[G \to .E, \$], [E \to .E+T, \$/+], [E \to .T, \$/+],$ $[T \to .T*F, \$/+/*], [T \to .F, \$/+/*],$ $[F \to .(E), \$/+/*], [F \to .id, \$/+/*]\}$
0E	1	$\{[G \to E., \$], [E \to E.+T, \$], [E \to E.+T, +]\}$	closed
0T	2	$\{[E \to T., \$/+], [T \to T.*F, \$/+/*]\}$	closed
0F	3	$\{[T \to F., \$/+/*]\}$	closed
0(	4	$\{[F \to (.E), \$/+/*]\}$ closed $\cup$	$\{[F \to (.E), \$/+/*/)], [E \to .E+T,)/+], [E \to .T,)/+],$ $[T \to .T*F,)/+/*], [T \to .F,)/+/*], [F \to .id,)/+/*]\}$
0id	5	$\{[F \to id., \$/+/*]\}$	
1+	6	$\{[E \to E+.T, \$/+]\}$ $\cup$	$\{[T \to .T*F, \$/+/*], [T \to .F, \$/+/*], [F \to .(E), \$/+/*], [F \to .id, \$/+/*]\}$
2*	7	$\{[T \to T*.F, \$/+/*]\}$ $\cup$	$\{[F \to .(E), \$/+/*], [F \to .id, \$/+/*]\}$
4E	8	$\{[F \to (E.), \$/+/*], [E \to E.+T,)/+]\}$	closed
4T		$\{[E \to T.,)/+], [T \to T.*F,)/+/*]\}$	which can be combined with 2... and so on...

3.7 General Remarks.

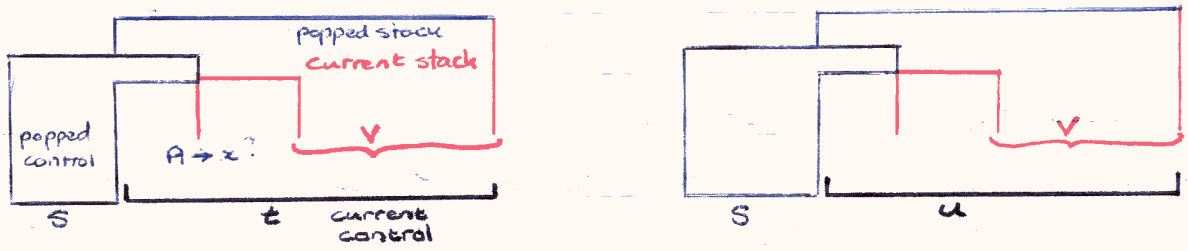
LR parsers are more powerful than LL, and can be automatically produced; in terms of power,

$LR(1) > LALR(1) > SLR(1) > LR(0)$.

Given a k, it is decidable whether a grammar is $LR(k)$, but if k is not given it is undecidable whether a grammar is $LR(k)$

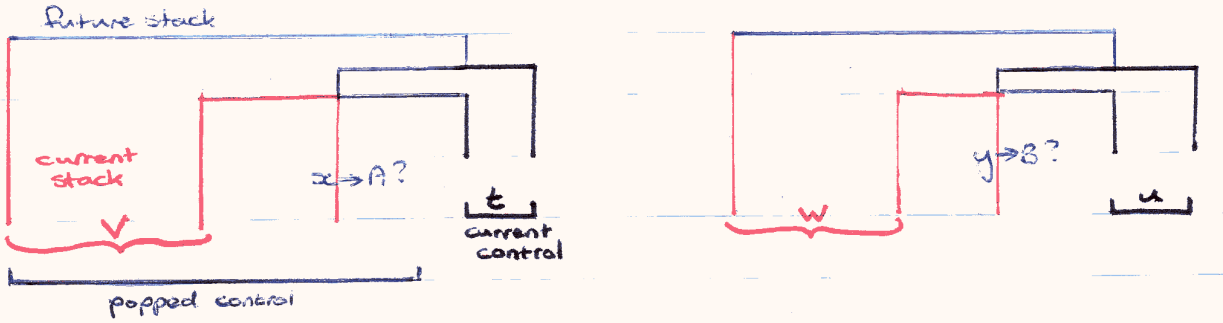
Under what conditions is a grammar LL or LR? To consider this, imagine a 'snapshot' of the parser at some moment in time

$LL(k)$ condition for determinism: if $t[1..k] = u[1..k]$ then $x = y$
(i.e. the first k symbols of the current control determine/decide the production)

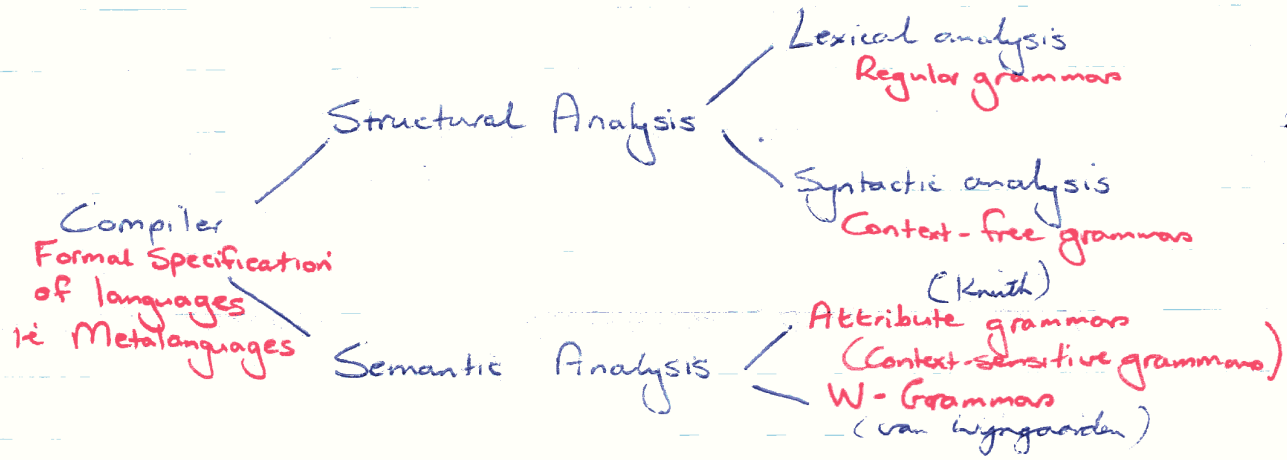


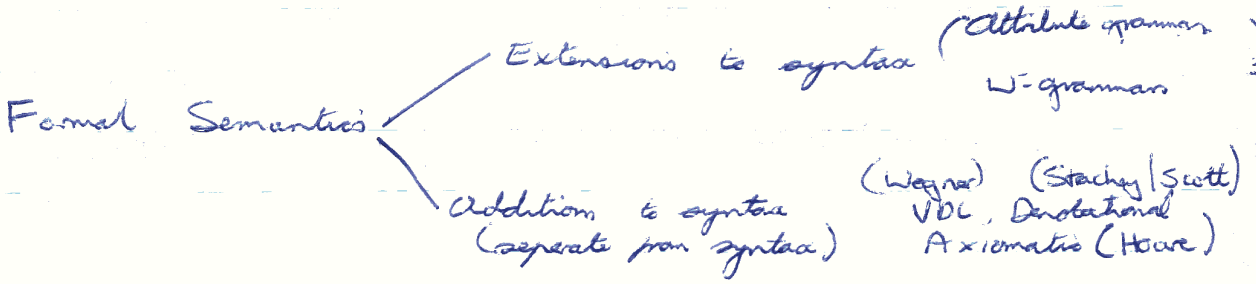
$LR(k)$ condition for determinism: if $Vxt[1..k] = Wyu[1..k]$ then

$V = W$, $x = y$, $A = B$



General Overview





"Semantics" are what happens when we interpret (we consider compiling as the first step of interpreting, namely interpreting the static semantics, while execution is concerned with the dynamic semantics

4 ATTRIBUTE GRAMMARS. ('SEMANTIC ANALYSIS')

Pass: name analysis
type checking
. types
. computable
. decidable

Semantic analysis cannot be done using a context-free grammar as we need to check across long distances. Using context-free grammars however, enables us to label our nodes meaningfully.
One way to get around this problem is using attribute grammars. Every non-terminal is given a set of attributes. This can deal with all grammars*. This can be compared to a compiler using a symbol table and displays (although this would be a useless formalisation of semantics).
The syntax grammar is extended to contain the attributes of each node (non-terminal) in the parse tree. These attributes are passed up and down the tree in a fashion determined by export and import lists.

Pagan With each distinct symbol of the context-free grammar there is associated a finite set of attributes, which notationally, are just names. With each distinct attribute there is associated a domain of values. A given attribute may be associated with any number of grammatical symbols. Each node of the syntax tree of a valid program is labeled not only with its grammatical symbol but also by a set of attribute-value pairs, one for each attribute associated with the symbol, and possibly by a logical condition expressing a constraint that must be satisfied by the attribute values involved. The value associated with an attribute occurrence in the tree is determined by various evaluation rules associated with the grammar's production rules.

■ Eg Suppose we are defining a machine-dependent language
* more powerful than CS; can handle Turing Machines)

for use on a computer with 32-bit words. An unsigned integer constant is thus syntactically invalid if it exceeds $2^{31} - 1$. While the set of all unsigned numbers is easily defined in BNF as:

<numeral> ::= <digit> | <numeral> <digit>

<digit> ::= 0 | 1 | 2 | 3 | ... | 9

it is extremely difficult to define the set of numerals $\{0, 1, \ldots, 2147483647\}$ concisely in BNF or its variants.

Instead we associate an attribute Val, corresponding to the domain of integers, and write the following specification:

<numeral>$_1$ ::= <digit> Val(<numeral>$_1$) ← Val(<digit>)

| <numeral>$_2$ <digit>

Val(<numeral>$_1$) ← 10 * Val(<numeral>$_2$) + Val(<digit>)

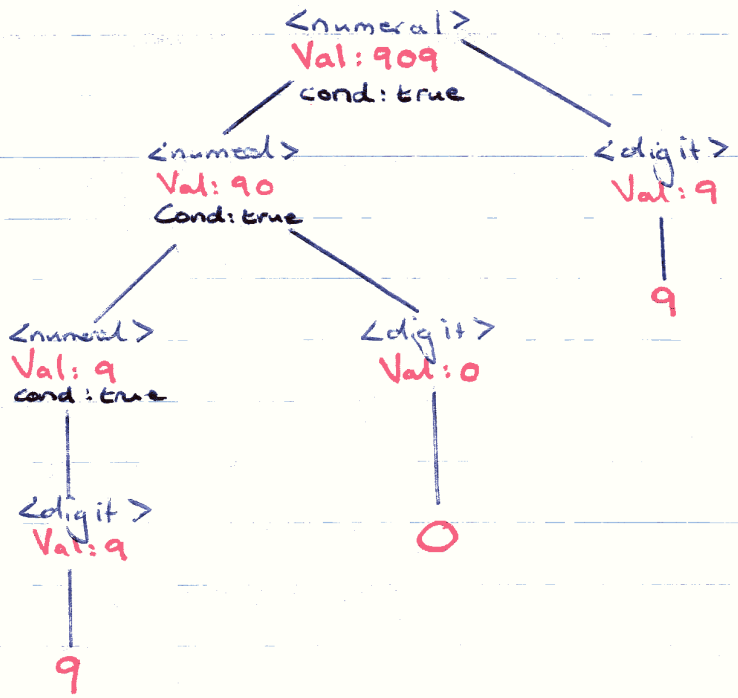
Condition: Val(<numeral>$_1$) ≤ $2^{31} - 1$

<digit> ::= 0 Val(<digit>) ← 0

| ...

| 9 Val(<digit>) ← 9

Eg parse tree for 909



In this example, we have information moving up the tree from the leaves to the root. We say that Val is a synthesised attribute of <numeral> and <digit>. It is possible for an attribute value at a node to be obtained from the node's parent; in this case we call it an inherited attribute. In general, a given grammatical symbol may have both synthesised and inherited attributes, and a given attribute may be synthesised with respect to one symbol and inherited with respect to another. Intuitively, a synthesised attribute at a node corresponds to information arising from the internal constituents of that construct, while an inherited attribute corresponds to information arising from the external context of the construct.

Under each alternative of a production rule, there must be an evaluation rule for each <u>synthesised</u> attribute of the symbol on the <u>left</u> (the symbol being defined) and for <u>each</u> <u>inherited</u> attribute of <u>each</u> symbol on the <u>right</u> (each symbol in the alternative)

In general we will have to deal with structured, non-numeric domains of attribute values. For our purposes we will make use of value domains that are enumerations, sets, tuples or sequences, according to the following conventions:

- the constants of an enumeration domain are arbitrarily chosen names enclosed in single quotes, eg 'proc'.
- for sets, the evaluation rules will employ the usual notation $\{\}$, $\cup$, $\in$ etc
- tuples belonging to a given domain consist of a fixed number of values, possibly of different types, in a certain order. The field selector operation is ., tuples are enclosed in parentheses () with fields separated by commas.

- a sequence is an ordered collection of any number of values of the same type. The notation $\langle \ldots, \ldots, \ldots \rangle$ denotes a sequence, with $\langle\rangle$ the empty sequence. The following primitive functions apply to sequences:

append (s, v)	first (s) (cf car)
concat ($s_1, s_2, \ldots, s_n$)	last (s)
length (s)	tail (s) (cf cdr)
	allbutlast (s)

For the sake of uniformity, we further decree that the specification of a subject language by means of an attribute grammar will consist of four parts:

- attributes and values (a list of attributes & domains)
- attributes associated with non-terminals
- production and attribute evaluation rules
- definition of auxiliary evaluation functions.

McD: Refer to the attribute grammar for Eva in notes.

An attribute grammar is a context-free grammar $G = (T, N, S, P)$ $V = T \cup N$, augmented by attributes $A = \bigcup_{x \in V} A_x$ where $A_x = I_x \cup S_x$ and $I_x \cap S_x = \Phi$. (i.e. A_x a disjoint union of inherited and synthesized attributes).
Note that if G is reduced, S appears only on the lhs of a production, so $I_S = \Phi$. Also $\forall x \in T$, $I_x = \Phi$.

We write a production $p \in P$ in the form $X_0 \to X_1 X_2 \ldots X_{n_p}$ where $n_p > 0$, $X_0 \in N$, $X_k \in V$.
For each occurrence X_k of a symbol X in p, for each attribute $a \in A_x$, there is an attribute occurrence a_k (cf k^{th} occurrence of X with defns / calls of procedures cf attribute occurrences with params (formal / actual), input of inherited cf attribute domains, output of synthesized

For each occurrence s_0 of a synthesized attribute on the LHS of a production p there is a semantic function $f^p_{s_0}$ (evaluation function) which yields the value of s_0 as a function of other attribute-occurences in p. (values move up the tree ; formal output)

For each occurrence i_k of an inherited attribute on the RHS of a production p $(1 \le k \le n_p)$ there is a semantic (evaluation) function which yields the value of i_k as a function of other attribute occurences in p. (values thus move DOWN the tree ; actual input)

We must avoid circular definition of attribute-occurence values, ie, we need a well-defined attribute grammar (WAG). Ideally we want to go from left to right in each production; inheritance in a DOWN traverse of the tree followed by synthesis in an UP traverse.

ALGORITHM DESIRED:

```
Processnode (X_0)                    {for any X examine its p}
    begin
        for k = 1 to n_p do
            if X_k ∈ N then begin
                update {i_k}
                processnode (X_k)
            end
        update {s_0}
    end
```

Initiate by calling Processnode (S)

An attribute grammar is

- COMPLETE if $\forall X \in V$ S_X is defined in all productions $X \to \ldots$
  I_X is defined in all productions $\ldots \to \ldots X \ldots$

- WELL-DEFINED (WAG) if complete, and with acyclic dependency graphs for all generable parse trees
  (i.e. all attributes are effectively computable)

- PARTITIONED if $\forall X \in V$, the attribute set A_X can be evaluated in the order $A_X^1, A_X^2, \ldots, A_X^m$ where:

  all disjoint $\begin{cases} A_X^i \subseteq S_X & \text{for } i = m, m-2, m-4, \ldots \\ A_X^i \subseteq I_X & \text{for } i = m-1, m-3, m-5, \ldots \end{cases}$

  and the dependency graphs for all productions are acyclic. (Every production has an associated evaluation algorithm 'attached' to every node where that production operates.)

- ORDERED (OAG) if it partitions as follows:

  $A_X^i = T_X^{m-i+1} - T_X^{m-i-1}$ where m is the smallest number such that $\underbrace{T_X^{m-1}}_{\text{synth.}} \cup \underbrace{T_X^m}_{\text{inh.}} = A_X$, where

  $T_X^{-1} = T_X^0 = \Phi$.

  $T_X^k = \{ a_X :$ b_X indirectly (pessimistically) depends on a_X implies $b_X \in T_X^j$ where $j \leq k \}$

  where a is synthesized / inherited for k odd / even.

Ordered attributes can be partitioned deterministically (look at actual dependencies, and produce all potential dependencies, and thus construct the total (pessimistic) dependencies)

We now consider some aspects of AG's.

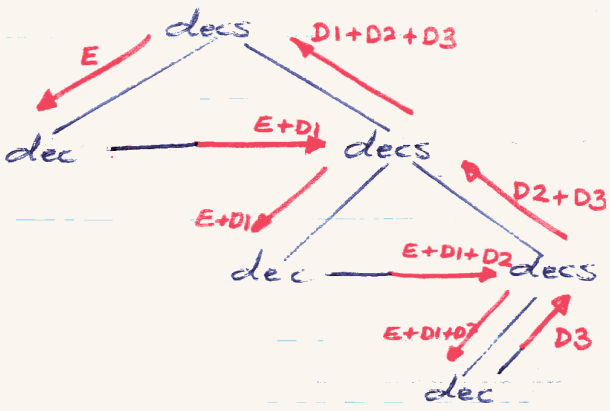
EVALUATION ORDER

In the AG for Eva, we have the following production.

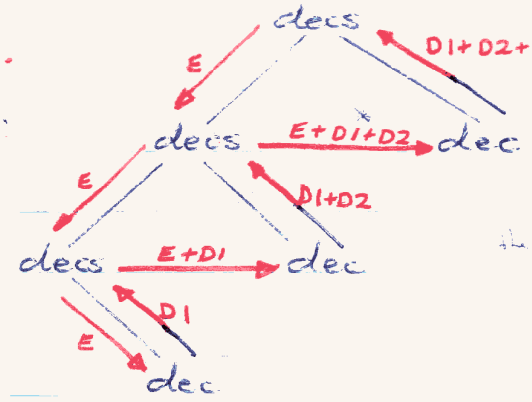
decs_E_d1+d2 → dec_e_D1 decs_e+d1_D2 (inherited / synthesised)

We can evaluate this in different ways. eg:

RIGHT RECURSIVE decs → dec decs



LEFT RECURSIVE decs → decs dec



STEPWISE decs_E1_e2+d2_d1+d2 → decs_e1_E2_D1 dec_e2_D2

synth
inher

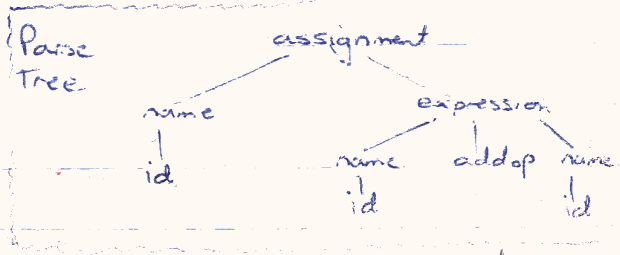
Consider the following attribute grammar

- assignment_E → name_e_INT_int expression_e_T_int
  | name_e_T1_t1 expression_e_T2_real

- expression_E_ cvt(T2, int) ∧ cvt(T3, int) → int, real* : t_T1
  → name_e_T2_t addop_t_O name_e_T3_t cvt(t, t1)

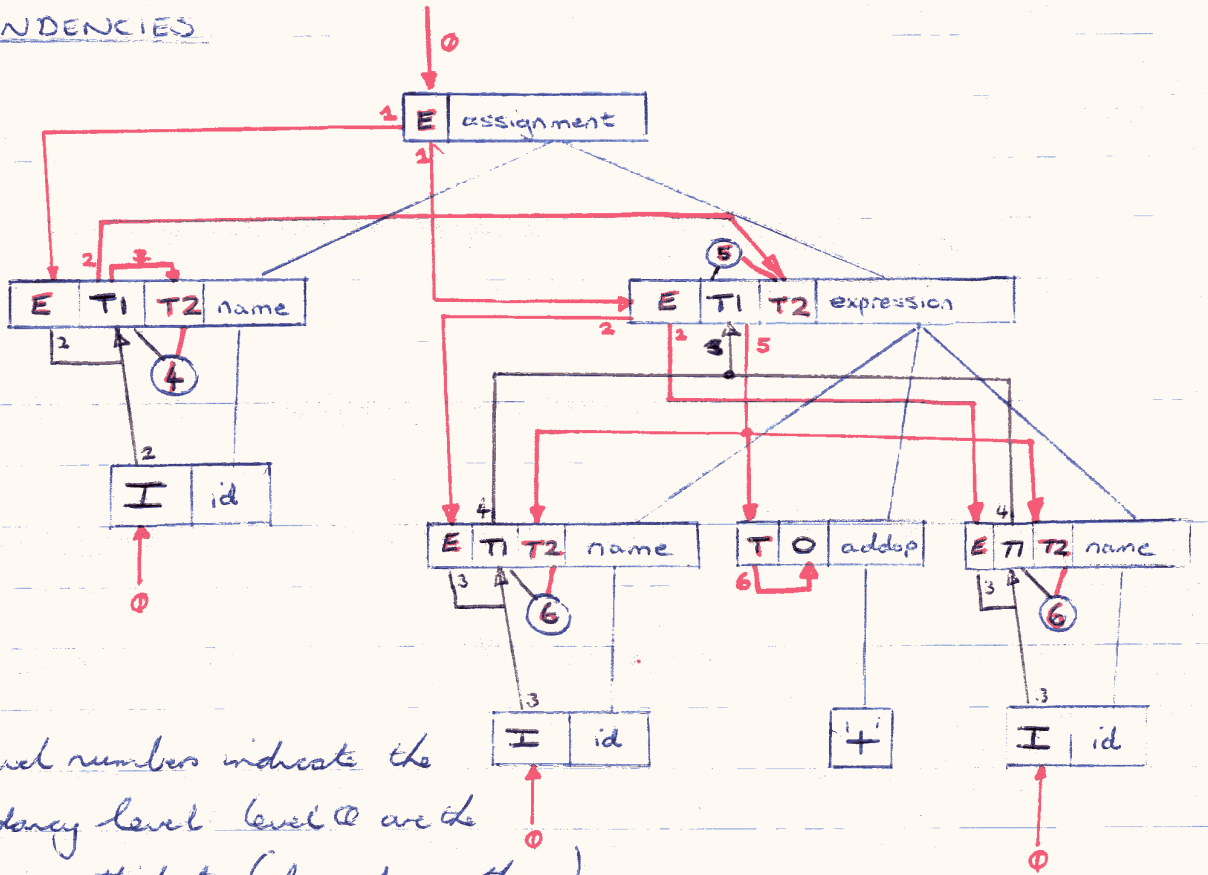
- name_E_ tp(i,e) : t_T1 → id_I cvt(t, t1)

- addop_INT_addint → '+'
- addop_REAL_addreal → '+'



* if T3 and T2 can be converted to int, answer is int, else real

DEPENDENCIES

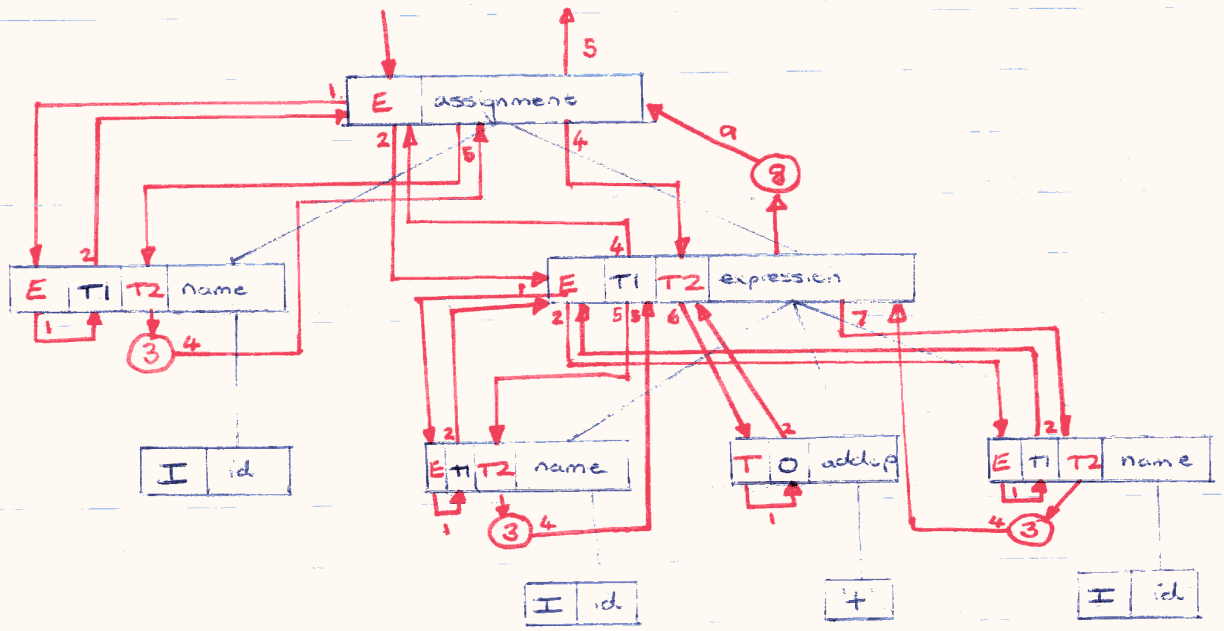


The level numbers indicate the dependency level. Level 0 are the intrinsic attributes (depend on nothing)

This dependency diagram shows us two things: (how do we do this for all trees?)

- our grammar (tree) is well-defined: no circular dependencies
- certain attribute calculations can be done in parallel - we have partitioned the evaluation into 7 stages (0-6). By finding these partitions we can in general detect parallelism.

Evaluation Order



Partition Sets (cf p32)

Eval order	Expression	Name	Addop	
1	{E}	{E}	{T}	inher
2	{T_1}	{T_1}	{0}	synth
3	{T_2}	{T_2}		inher
4	{•}	{•}		synth

conditions

The partitions always alternate between inherited and synthesised, and always start on inherited and end on synthesised attributes.

Ordered Attribute Grammars

In order to check that the grammar as a whole has no cycles we need to be totally pessimistic. If we can partition the attributes in a grammar into subsets which are evaluated sequentially in order, with the odd partitions inherited and the even ones synthesised the grammar is ordered (not all well-defined AG's are ordered). We can consider these partitions

to be sets of co-routines, and can say that if we can create co-routines for the gramma. then it is ordered.

IMPLEMENTATION CONSIDERATIONS (figure refs are to Watt & Goose)

Evaluation functions can be co-routines attached to each node through pointer pointing to the routine's re-entrant code.

- simple translation of code in Simula (Fig 8.16)
- recursive procedures using case statement and parameter passing (Figs 8.17 & 8.18)
- using tables and a table driver evaluation function.

Attribute storage:

Intermediate attributes:

- overlaying of attributes (i.e store only those which are necessary; only set aside fixed storage for the final attributes)
- use of local variables of the evaluation procedure

Final attributes

- storage only at specified nodes (pointers at other nodes referring to it)
- global storage (eg at root)

Refs SIGPLAN V14N2? 1979 DAVID A. WATT - "AN EXTENDED AG FOR PASCAL"

Acta Informatica V13 P229-256 Uwe Kastens - "Ordered Attributed Grammars"

Types of Attribute Grammars.

A list of classes of AG's is given in descending order of expressive power (i.e. increasing restrictions on dependencies)

AG		Knuth 1968
WAG	well-defined	"
ANCAG	absolutely non-circular	Kennedy & Warren 1976
OAG	ordered	Kastens 1980
m-APAG	m-alternating passes	Jazayeri & Walter 1975
n-PAG	n-left to right passes	Bochmann 1976
1-PAG	1-left to right pass	Bochmann 1976
= L-AG	L-attributed	Lewis, Rosencrantz & Stearns 1974
S-AG	S-attributed	Lewis, Rosencrantz & Stearns 1974

5 W-GRAMMARS (2-level Grammars)

Pagan

W-Grammars were introduced by Van Wijngaarden for the Algol-68 report. They are capable with dealing with context-sensitive aspects of a language

We just describe a metalanguage for context-free grammars, and then extend these to W-grammar

A protonotion is any sequence of lowercase letters and spaces where the spaces are ignored (i.e. something and some thing are the same protonotion). A protonotion is a notion of a particular grammar if there is a production rule that defines it, so that "notion" is basically a new word for "nonterminal". A protonotion ending in symbol corresponds to a terminal symbol of the subject language; in order to eliminate all possible conflicts between metalanguage and subject language, such protonotions are always used in productions, while the correspondence between these and the actual symbols of the language are listed in a separate representation table.

There are four punctuation metasymbols; the left and right sides of a rule are separated by a colon; alternative definitions are separated by semicolons; the protonotions are separated by commas, and the entire rule is terminated by a period.

A hyper-rule is an abstraction of a number of different production rules sharing a common pattern. For example, the BNF grammar for EVA contains three production rules which we may now rewrite as:

declaration sequence : declaration ; declaration sequence, declaration.

statement sequence : statement ; statement sequence, statement.

letter sequence : letter ; letter sequence, letter.

All these rules can be replaced by the hyper-rule

SEQITEM sequence : SEQITEM ; SEQITEM sequence, SEQITEM.

where the use of upper-case letters signifies that SEQ ITEM is a metanotion which stands for any number of protonotions. The latter are specified by a metaproduction rule or metarule; with the metanotion as its left side. This particular metanotion could be defined by the metarule

SEQ ITEM :: declaration ; statement ; letter.

The production rules represented by a given hyper-rule are obtained by consistent substitution of protonotions (specified by the metarules) for the contained metanotions

The RHS of a metarule may also involve metanotions, possibly the metanotion being defined. By convention, metanotions are not broken up by spaces, spaces are used to separate meta/protonotions in a metarule

Apart from the representation table, a two level grammar consists of a set of metarules and a set of hyperrules. Each hyperrule stands for a number of production rules, possibly an infinite number, but only one if it involves no metanotions. For each metanotion used, there must be a metarule to specify what protonotions it represents. The metarules may be thought of as a higher-level grammar that (with the aid of hyperrules) generates a grammar, where the generated grammar may involve an infinite number of notions and have an infinite number of production rules (or alternatives thereof). This is basically the reason why 2-level grammars are fundamentally more powerful than BNF variants which are finite.

To illustrate the ability of 2-level grammars to handle context sensitive aspects of languages, consider the

grammar $x^n y^n x^n$, which cannot be defined by a context-free grammar.

We use two metanotions:

LETTER :: letter x; letter y; letter z.

TALLY :: i; TALLY i.

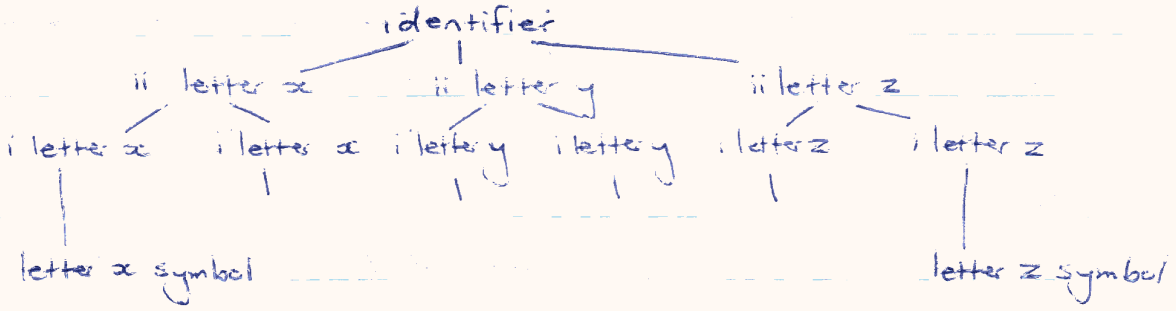
and the hypernotion:

identifier : TALLY letter x, TALLY letter y, TALLY letter z.

TALLY i LETTER : TALLY LETTER, i LETTER.

i LETTER LETTER symbol.

Tree for x x y y x x



A 2-level grammar can be conceived of as a 2-level macro builder which builds up a tree containing expressions of the non-terminals. Parsing such grammars requires a pattern-matching algorithm. W-grammars are very precise but not very practical.

We will now consider aspects of a 2-level grammar for Eva.

program : new block containing DECS, where DECS consistent.

- A program is a new block containing consistent DECS

DECS :: DEC ; DECS DEC. DEC :: TYPE type TAG.

TYPE :: VALTYPE ; proc ; proc with PARAMETERS.

TAG :: LETTER ; TAG LETTER. LETTER :: letter ALPHA

ALPHA :: a; b; c; ... x; y; z. VALTYPE :: char; string

PARAMETERS :: PARAMETER ; PARAMETERS PARAMETER.

PARAMETER :: VALTYPE type TAG.

- We arrange for most of the grammar notions to each begin with a protonotion represented by the notation NEST, defined by:

  NEST :: new ; new DECS ; NEST new DECS.

  A notion of the form NEST NOTION will characterise a syntactic unit of an Ever program iff the NEST protonotion records the name and type information specified by just those declaration sequences and/or parameter lists which bear upon that syntactic unit. Thus the outer block will have its dummy next new. The notion for each principal construct inside the main block will have a next of the form new new DECS ; the next inner block a next of the form new new DECS1 new DECS2, etc.

- Note that complete information on the parameters of a procedure, if any, is included in the protonotion corresponding to a procedure name. For example, the protonotion:

  proc with string type letter s char type letter c letter c type letter s letter a letter b ....

  corresponds to proc sub (string s, char cc) = ...

- In the case of the block and declaration constructs, we embellish our syntactic notions still further by supplying a DECS protonotion corresponding to the new set of declarations involved. Where we would simply have the notion block in a context-free grammar, we will now be dealing with notions of the form NEST block containing DECS, where the DECS part records the information given by the declarations inside the block. For example, the "topmost" hyperrule in the

grammar is:

Program : new block containing DECS, where DECS consistent.

The predicate part where DECS consistent corresponds to the context condition that states that a name cannot be declared more than once at the same level in a program. The formalisation of this condition requires a number of auxiliary hyperrules. First, if the DECS is actually a single DEC, there is no problem:

Where DEC consistent : EMPTY.

Otherwise, according to the metarules, the DECS must be of the form DECS DEC, which is the same as DECS TYPE type TAG. Since the type information is not relevant here, we merely stipulate that the final TAG must not appear earlier in the protonotion, as well as (recursively) requiring the earlier part to be itself consistent.

Where DECS TYPE type TAG consistent :
    Where DECS consistent, Where TAG not in DECS.

Where TAG not in DECS DEC : Where TAG not in DECS,
    Where TAG not in DEC.

Where TAG1 not in TYPE TAG2 : Where TAG1 is not TAG2.

EMPTY : .

- if the second level grammar is finite then metarules and hyperrules simply abbreviate many production rules
- if the second level grammar is itself context free then the generated grammar can have an infinite number of notions and productions and can be of type 2 (see PTO notes)

Eg

LETTER	::	x ; y ; z .	finite choice ⇒ abbreviation
N	::	1 , N1 .	infinite language
identifier	::	Nx , Ny . Nz .	infinite alternatives on RHS
N1 LETTER	:	N LETTER , 1 LETTER .	infinite number of rules
LETTER	:	LETTER symbol	finite number of rules ⇒ abbreviation

6. THE SECD MACHINE (an operational semantics)

LANDIN Comp. Journal 64 p308

Operational semantics defines semantics by setting up a abstract machine for interpreting the language; meaning is therefore considered to be analagous to value

McCarthy's LISP language runs on the abstract SECD machine. As LISP is sufficiently powerful to allow other languages to be written in it; the SECD machine can be considered to provide a general programming semantics method of description. The SECD machine is based on the lambda calculus, a calculus of functions (all values are functions).

SYNTAX OF LAMBDA EXPRESSIONS

<exp> ::= <id> | <application> | <abstraction>
<application> ::= γ <operator> <operand>
<operator> ::= <exp>
<operand> ::= <exp>
<abstraction> ::= λ <bound var> <body>
<bound var> ::= <letter>
<body> ::= <exp>

McCARTHY CONDITIONS

We are used to conditions of the form
if <boolean-exp$_1$> then <exp$_1$>
else if <boolean-exp$_2$> then <exp$_2$>

In this formalism, we replace these by:
<boolean-exp$_1$> → <exp$_1$>, <boolean-exp$_2$> → <exp$_2$>
eg zero(x) → 1, t → x * f(x-1)
(effectively just zero(x) → 1, x * f(x-1) cf cond in LISP)

The SECD machine consists of four components: the Stack (evaluations), Environment (store), Control (input expression) and Dump (procedure calls).

The evaluate function could be defined as:

evaluate (E, x) = is-id (x) → E (x),
is-application (x) → operator(x) (operand(x)),
is-abstraction (x) →
λ u. evaluate (extend (E, u, var (x)), body(x))
where extend (E, u, x) = λy. y = x → u, E(y)

In the description of the machine, the letter h will be used to stand for head, and t for tail, so, for example
hS is the head of the stack (cf car, cdr)

The Lispkit interpreter can be considered as a machine which takes symbolic expressions as input, evaluates them, and outputs the values of the S-exprs. The output function could be described as follows

output (E, x) = iterate (transform, <empty, E, x, empty>)
where
iterate (F, <S, E, C, D>) = null (C) ∧ null (D) → S,
where iterate (F, F(<S, E, C, D>))
transform (<S, E, C, D>) =
null (C) → <hS - S', E', C', D'> where <S', E', C', D'> - hD,
is-id (hC) → <E (hC) - S, E, tC, D>,
is-application (hC) → <S, E, rand (hC) - rator (hC) - Y - tC, D>,
is-abstraction (hC) → <closure (hC, E) - S, E, tC, D>,
hC = 'Y' →
is-closure (hS) → <empty, extend (env (hS), hthS, var (hS)),
body (hS), <thS, E, C, D>>,
<hS (hthS) - ththS, E, tC, D>

LANDIN Communications of the ACM 64 P308

Note the use of functionals to give the meaning of functionals.
The program is part of the data structure (state vector) <SECD>
The interpreter is effectively a function from string symbols to interpretation.

McCarthy's Work in the Field of Computability

McCarthy formulated a list of problems with which the emergent theory of computation should concern itself, as well as a set of methods and a formalism for dealing with those, which is capable of dealing with symbolic rather than numeric data. Since his formalism necessitated the use of recursion, he investigated the extent to which recursive programs and iterative programs were equivalent. He also developed the technique of recursive induction for proving pairs of recursive programs equivalent.
McCarthy applied his formalism to semantics with the idea that that the meaning of a program can be expressed in terms of the change it effects on the state vector of an idealised interpreter.
McCarthy's formalism is particularly attractive as it has the form of a high level language, and can thus be used to explain its own semantics as well as being far more useful practically than any other such formalisms (eg FSMs).
M.'s formalism is based on an idea similar to that of general recursive functions: there is a set F of primitive functions and a set C of functionals for building new functions out of the old, the claim being that the closure C(F) comprises all computable functions.
Actually, McCarthy just suggested a set C_{mc} of strategies, leaving the user to define a base set F appropriate to the

From 'The Theory of Computer Science' by J M Brady, Chapman & Hall, London 1977
DE001.6401 BRAD

data structures. In the following section, we define S-expressions and a corresponding set FUNCTIONS of functions. Give rules of generalized composition, conditional expressions, and recursion. Composition generalizes function nesting and provides a form of sequencing.

McCARTHY CONDITIONALS

General form $(p_1 \rightarrow e_1, p_2 \rightarrow e_2, \ldots, p_n \rightarrow e_n, e_{n+1})$
which corresponds to the ALGOL 60 expression

if p_1 then e_1
else if p_2 then e_2
else if ...
else if p_n then e_n else e_{n+1}

(McCarthy actually suggested the incorporation of conditional into ALGOL in 1959.)
We introduced conditional expressions as a piece of formal mathematical notation, and then gave a process interpretation. However, in contemporary mathematics an expression merely has a value; it is not part of mathematics to say how that value is computed. Purists could object to an evaluation process rule in the role of a definition of a piece of mathematical notation. Note that there cannot be a simple function $f(p_1, e_1, \ldots, p_n, e_n)$ for computing the value of conditionals, since $(1>2 \rightarrow 3/0, t \rightarrow 4)$ has the value 4, but $f(1>2, 3/0, t, 4)$ is undefined. We can use lambda abstraction to 'delay' the evaluation of the e_i's to get around this. In particular, $(p_1 \rightarrow e_1, t \rightarrow e_2)$ can be modelled by:

$$[\text{ifasfn}\ (p_1)\ (\lambda().e_1, \lambda().e_2)]()$$

where ifasfn ("if as a function") is a function that returns a second argument selector when applied to false, and a first argument selector when applied to true.

Nevertheless, the conditional expression fundamentally involves understanding a process, and thus could be a suggestion of the inadequacy of modern mathematics which does not incorporate the idea of process

NB: lambda notation (Church) should not be confused with lambda calculus (Church 1941). Lambda notation is a way of giving a systematically complicated default name to any function (eg $\lambda(x,y) \cdot x^2+y$ which is the name of the function which computes x^2+y) lambda calculus is a set of conversion rules allowing λ-names to be changed. The conversion rules reflect the operations of renaming bound variables, applying a function to a set of arguments, etc. Church argued that λ-calculus embodied a precise definition of 'computable'; it has the same power as Turing machines

We can use λ-notation to write our functions. Consider a function

$$abs(x) = (x \geq 0 \to x,\ t \to -x)$$

We apply λ-abstraction, giving the default name:

$$abs = \lambda x.\ (x \geq 0 \to x,\ t \to -x)$$

There is a problem in applying λ-abstraction to recursive definitions such as:

$$factorial(x) = (x = 0 \to 1,\ x * factorial(x-1))$$

because of the definition containing its own name (whereas in λ-notation, the definition is the name). However, considering the general form of a recursive definition

$$f(x) = E(x, f)$$

By λ-abstraction, $f = \lambda x.\ E(x, f)$, where there may be any number of f's on the RHS. We reduce this to one by:

$$f = \{\lambda g.(\lambda x.\ E(x, g))\}(f)$$

Now, $\lambda g.\ \lambda x.\ E(x,g)$ is a functional (say F), which takes f as argument. We have: $f = F(f)$, so that f is a <u>fixed point</u> of F. We now show that there is a function Y (with a complex λ-name) which, when applied to F, yields one of its fixed points f. i.e. $Y(F) = f$, so that $Y(\lambda g.\ \lambda x.\ E(x,g))$ is the λ-name for f.

We want to find f so that $F(f) = f$. This is solved by:

$f = \text{selfapply}(h)$ where $h = \lambda y.\ F(y(y))$

since
$$\begin{aligned} f &= h(h) && \text{(defn of selfapply)} \\ &= \{\lambda y.\ F(y(y))\}(h) && \text{(defn of } h\text{)} \\ &= F(h(h)) && \text{(subst. for } y\text{)} \\ &= F(f) && \text{(since } f = h(h)\text{)} \end{aligned}$$

Now we can derive a λ-name for Y. selfapply has a name $\lambda g.\ g(g)$, so the full λ-name for f is
$$\{\lambda g.\ g(g)\}(\lambda y.\ F(y(y)))$$

Now we have $Y(F) = f$, so that finally:

$$Y = \lambda G.\ \{\lambda g.\ g(g)\}(\lambda y.\ G(y(y)))$$

Y was invented by Curry and Feys (1953) and is called the paradoxical combinator (since selfapply generates a number of paradoxes). The development of Y has a number of interesting theoretical consequences; it is not suggested as a practical mechanism for evaluating recursively defined function applications!

S-EXPRESSIONS

Sexprs are essentially binary trees with identifiers (atoms) for their leaves. von Neumann architecture, using contiguous sequential memory, is inappropriate for computing with symbolic structures. In particular

- symbolic data structures can become unpredictably large (storage must be allocated incrementally)
- symbolic processing general calls for the replacement of one structure with another of a different size.

These problems resulted in a set of techniques called (collectively) list processing.

The syntax of S-expressions is

<sexpr> ::= <atom> | (<sexpr> . <sexpr>) | <list> | <liststructure> (dotted pair)

<list> ::= ({<atom>}$_0^\infty$)

<liststructure> ::= ({<sexpr>}$_0^\infty$)

<list>s and <liststructure>s are in fact a special case of dotted pairs, useful as shorthand. (see later)

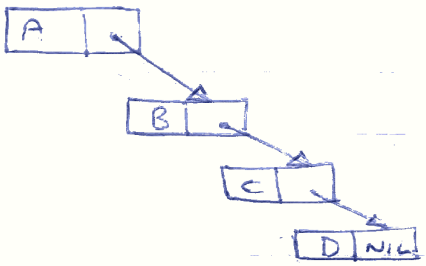
We now describe FLISP. The set of computable functions of Sexprs is thus Cmc(FLISP) (the set of LISP computable functions).

- predicate atom : Sexpr → {t, nil}
- predicate eq : Sexpr × Sexpr → {t, nil, ⊥} (defined for atoms only, cf nodes & subtrees)
- selectors car : Sexpr → Sexpr left subtree } undefined for atoms
  cdr : Sexpr → Sexpr right subtree
- constructor cons : Sexpr × Sexpr → Sexpr builds tree

List Notation.

To describe binary trees we need an ordered data structure. Lists describe linear binary trees. We introduce the empty tree NIL to make relationships generally applicable even to the last elt of a list (as $c^a r$ (atom) is undefined, but $c^a dr$ (atom, nil) is not!)

Eg (A.(B.(C.(D.NIL)))) (≡ (A B C D))



The following recursive function list-to-pair describes a process for converting list notation into the corresponding longhand dotted-pair as follows.

$$\text{list-to-pair}((e_1 \ldots e_n)) = (n=0 \to \text{nil},\ t \to e_1.\ \text{list-to-pair}((e_2 \ldots e_n)))$$

Application of the Formalism: Programming Semantics

In order to give the semantics of a programming language L, we need a precise way of specifying well-formed programs PROG_L in L (ie syntax). Secondly, we need a set M_L of possible meanings of programs in L, so that if $P \in \text{PROG}_L$, then the meaning of $P \in M_L$. Finally, we need some way of associating with any program P its meaning. We would like our programs to be deterministic, which suggests modelling the meaning of L

by a function L-MEANS : $PROG_L \rightarrow M_L$

We presumably need to insist that L-MEANS should be computable; however, our determinism requirement is not completely general, and we must consider L-MEANS being a function as a special case.

$PROG_L$

The meaning of a program is to a large extent independent of the particular (strings of) symbols chosen to represent its syntactic constructs. However there is generally a strong relation between grammatical structure and semantics. McCarthy observed that an alternative formal specification for syntax of programs more appropriate for semantics can be created from selectors and predicates. The predicates correspond to the distinct syntactic structures, (eg is-id?), the selectors 'parse' these by selecting components of non-terminals. This method of specifying syntax has been taken up in VDL. McCarthy called such a syntactic specification <u>abstract</u> and <u>analytic</u> in contrast with <u>concrete</u> (since it does not insist on a particular external representation) and <u>synthetic</u> (since it details the analysis of programs rather than their construction or synthesis)

eg is-prog (P) = is-const (P) ∨ is-vble (P)
∨ (is-sum (P) ∧ is-prog (left (P))
∧ is-prog (right (P)))
∨ (is-product (P) ∧ is-prog (left (P))
∧ is-prog (right (P))

M_L

M_L is considered to be a set of functions from states to states

Eg Consider the language C of a simple one-address computer with a single accumulator. The commands are:

LOAD	<reg>	acc ← reg
STO	<reg>	(reg) ← (acc)
ADD	<reg>	acc ← (acc) + reg
LI	<const>	Load immediate acc ← c

Formal syntax

is-command (c) = (is-load(c) and is-reg(sel-reg(c)))
or (is-sto(c) and is-reg(sel-reg(c)))
or (is-add(c) and is-reg(sel-reg(c)))
or (is-lim(c) and is-val(sel-arg(c)))

A state is a pairing of registers R with values V. We define the function

update : R × V × S → S

where update(r, v, s) is the state resulting from updating s by associating value v with register r. The function lookup associates registers with their values

lookup : R × S → V

We define C-means in terms of the function step, which describes the result of executing one command

step(c, s) = (is-load(c) → update(acc, lookup(sel-reg(c), s), s)
is-sto(c) → update(sel-reg(c), lookup(acc, s), s),
is-add(c) → update(acc, lookup(acc, s) + lookup(sel-reg(c), s), s)
is-lim(c) → update(acc, sel-arg(c), s))

C-means(P) = λs. null(P) → s, C-means(rest(P))(step(first(P), s))

McCarthy's approach to semantics has been substantially developed, particularly by VD Lab. McCarthy and Painter (1967) prove the correctness of a compiler for a simple expression language E (cf bottom p53) into the machine language C. Their proof introduces the technique of structural induction, later developed by Burstall (1969)

THE METACIRCULAR EVALUATOR

The LISP evaluator embodies the environment model of evaluation, which has two basic parts:

- to evaluate a compound expression (other than a special form (eg define)) evaluate the subexpressions and then apply the value of the operator subexpression to the value of the operand subexpressions
- to apply a compound procedure to a set of arguments, evaluate the body of the procedure in a new environment, constructed by extending the environment part of the procedure object by a frame in which the formal parameters of the procedure are bound to the actual arguments to which the procedure is to be applied.

These two rules describe the essence of the evaluation process, a basic cycle in which expressions to be evaluated in environments are reduced to procedures to be applied to arguments, which in turn are reduced to new expressions to be evaluated in new environments, and so on, until expressions are reduced to symbols (whose values are looked up in the environment) and primitive procedures (which are applied directly)

Given the ability to apply primitives, we may question the need for the evaluator. The job of the evaluator is not so much to specify the primitives of the language as to

provide the means of combination and abstraction which bind the primitives to form a language. Specifically the evaluator:

- allows us to deal with nested expressions
- allows us to use variables
- allows us to define compound procedures
- provides the special forms which must be evaluated differently from procedure calls

The core of the evaluator consists of two procedures eval and apply. eval takes as arguments an expression and an environment. It classifies the expression and directs its evaluation. eval is structured as a case analysis of the syntactic type of the expression to be evaluated. Each type of expression has a predicate that test for it and selectors for accessing its parts. Eval is defined as:

```
(define (eval exp env)
  (cond
    ((self-evaluating? exp) exp)                    (eg numbers)
    ((quoted? exp) (text-of-quotation exp))
    ((variable? exp) (lookup-variable-value exp env))
    ((defn? exp) (eval-defn exp env))
    ((assignment? exp) (eval-assignment exp env))
    ((lambda? exp) (make-procedure exp env))
    ((conditional? exp) (eval-cond (clauses exp) env))
    ((application? exp) (apply (eval (operator exp) env)
                               (list-of-values (operands exp) env)))
    (else (error "Unknown-exp-type -- " exp))
  )
)
```

apply takes two inputs, a procedure and a list of arguments to which the procedure is to be applied.

```
(define (apply proc args)
  (cond
    ((primitive-proc? proc) (apply-primitive proc args))
    ((compound-proc? proc)
     (eval-sequence (proc-body proc)
                    (extend-environment
                      (parameters proc)
                      arguments
                      (proc-environment proc))))
    (else (error "unknown proc type" proc))))
```

The evaluator uses data abstraction to decouple the general rules of operation from the details of how expressions are represented. The syntax of the language being evaluated is thus solely determined by the procedures which classify and extract pieces of expressions (the predicates and selectors)

ENVIRONMENTS.

An environment is a sequence of frames, each of which is a table of bindings associating variables with values. A variable in an environment is looked up by finding the binding for the variable in the environment, and returning the value part of the binding. To extend an environment

by a new frame associating variables with values, we construct a new frame of bindings and adjoin this frame to the environment. To set a variable to a new value in a specified environment, we alter the value part of the binding. To define a variable, we adjoin a new binding to the head of the first frame in the environment, if the variable is already bound in that frame we can do exception processing. (eg bind new value; generate error, etc)
An environment can be represented as a list of frames, where a frame is a list of bindings, where a binding is a pair containing a variable and its value. (This is only one of many plausible representations for an environment)

TREATING EXPRESSIONS AS PROGRAMS

One operational view of the meaning of programs is that a program is a description of an abstract (perhaps infinitely large) machine. For example, consider the following program to compute factorials:

```
(define (factorial n)
  (if (= n 1)
      1
      (* (factorial (- n 1)) n)))
```

We can regard this program as a description of a machine containing parts that decrement, multiply, and test for equality, together with a two position switch and another factorial machine (the factorial machine is infinite as it contains another factorial machine within it)
In a similar way, we can regard the evaluator as a

machine which takes as input a description of a machine. Given this input, the evaluator configures itself to emulate the machine described. For example if we fed the evaluator with a description of the factorial machine, it would be able to compute factorials. The evaluator can thus be regarded as a universal machine capable of mimicing other machines when given their descriptions. (it thus regards the users 'programs' as its 'data', one of the reasons why this distinction is unclear in Lisp)

OVERVIEW OF LISP

Lisp is :

- an expression Language (S-exp.s)
- a functional Language (expression, command & binding structures implemented by functions)
- a symbol-processing language (symbolic atoms)
- a list-processing language (lists are S-exp.s)

The LISP machine is an already 'implicit' function EVAL to which successive LISP expressions are submitted, and interpreted as function applications to be evaluated.

DATA: linked lists of linked lists of... of atoms (symbolic or numeric) all referenceable through a central object list (OB-list). Each atom has a property list (p-list) attached, structured in (<attribute name>, <attribute value>) pairs.

PROGRAM: data list with prefix structure and atoms playing the role of identifiers. Sequencing is by functional composition, alternatives by COND function, looping by recursive function calls.

ENVIRONMENT - most recent association, stored at beginning of association list (a-list), with global associations (stored as 'APVAL') on p-list taking precedence. The a-list is accessible to programmers through setq. Parameters can be passed evaluated or unevaluated (quoted). User functions are of type EXPR or FEXPR, primitive functions are of type SUBR or FSUBR.

Other Readings

"A Formal Description of a Subset of Algol" J. McCarthy. Proc. IFIP, North-Holland, 1966.

OPERATIONAL SEMANTICS & SECD

We generalise Church's lambda expressions (in which even numbers are λ-exps) to get Landin's applicative expressions (AE's) in which we include numbers, (lists of) identifiers, λ functions, etc. Many aspects of high level languages can be directly carried over into AE's (including recursive functions using Curry's paradoxical combinator). Consider the conditional expression if p then a else b : We could describe this by {ifasfn (p)} (a, b) where ifasfn is a λ-function which returns a function first which selects a, or a function second which returns b. More precisely :

$$ifasfn(p) = \begin{cases} first \text{ where } first = \lambda(x,y).x & \text{if } p \text{ is true} \\ second \text{ where } second = \lambda(x,y).y & \text{if } p \text{ is false} \end{cases}$$

This satisfactorily handles many conditionals, but not those with the form if a=0 then b else (1/a). We have to delay the evaluation of both the conclusion and alternative. λ( ). 1/0 is a well defined function, even if an attempt to apply it results in an error. We can thus render our conditional more adequately by:

{(ifasfn (p)} (λ( ). a, λ( ). b)} ( )

The same technique can be used to give meaning to for commands, and call by name. P. Z. Ingerman (1961) invented a technique for implementing call by name based on procedures known as 'thunks'. This idea is useful in using AE's to accomplish the same. If we have the Algol declaration: integer procedure simple (x) ; integer x;

simple := x + 2;

the body is transformed into simple := x( ) + 2 and a call simple (a+b) transformed into simple(λ( ). a+b)

EVALUATING AE's

(This section is in essence equivalent to the SECD section).
The most direct approach to defining the values of AE's is to define a function evaluate(x) giving the value of the AE x. We consider the environment to be a function E giving the values of the free variables (identifiers) in x.
Thus we define:

val(E, x) = is-ident(x) → E(x),
is-comb(x) → {val(E, sel-opr(x))}[val(E, sel-opd(x))]
is-lambda(x) → f

where f(u) = val(Enew, sel-body(x))
where Enew = extend(E, u, sel-bv(x))

The function extend updates the environment of E by binding u to the bound variable part of x.

$$extend(E, u, x) = \lambda y.\ y = x \rightarrow u, E(y)$$

Whatever the merits of evaluate as a model for AE's, it is certainly not obviously related to the way AE's can be evaluated by machine. - how are AE's 'run'? Landin (1964) defined a set of states, and a function transform : State → State, such that repeated application of transform described mathematically a sequence of steps to evaluate an AE. A state (<S, E, C, D>) consists of a stack, environment, control and dump. (In terms of ALGOL60 runtime storage, the control is the code being executed, the stack S the current frame, the dump is the frame stack up to and including the previous frame and the environment is the static link or display)

```
transform (<S,E,C,D>) =
    null(C) → <hd(S) :: S', E', C', D'> where D = <S', E', C', D'>
  (let x = hd(C)
      is-ident(x) → <lookup(E,x) : S, E, tl(C), D>,
      is-comb(x) →
                  where Cnew = sel-opd(x) : sel-opr(x) : (ap :: tl(C))
      is-λexp(x) → <mk-closure(x,E) : S, E, tl(C), D>,
      x = ap →
          (let f = hd(S)
            not is-closure(f) → <Snew, E, tl(C), D>
                          where Snew = f(2nd(S)) : tl(tl(S))
           (let cl-body = sel-body(f)        /* body of closure */
            let cl-bv = sel-body(f)
            let cl-env = sel-env(f)
            <[], extend(cl-env, 2nd(S), cl-bv), cl-body, Dnew>
                where Dnew = <tl(tl(S)), E, tl(C), D>
          )))
```

To operate transform, we start with an empty stack and dump, the AE to be evaluated on control, and the initial environment as the environment component of the state. The function transform is repeatedly applied until the control and dump are empty. (see Landin, 1964). Most of the complexity of transform is in the application of closures (corresponding to starting a new runtime stack frame).

Eg evaluate sq(a) in environment [a.3]. Successive snapshots of the state are:

<[], [a.3], sq(a), []>
<[], [a.3], [a sq ap], []>
<[3], [a.3], [sq ap], []>
<[sq 3], [a.3], [ap], []>
<[9], [a.3], [], []>

Unlike evaluate, transform specifies the evaluation order: first the operand, then the operator, then apply the latter to the former.

Eg2 evaluate $U(\lambda x.y)$ in environment $[y.27]$
where $U = \lambda f.\{\lambda y.f(y)\}(2)$

$\langle [], [y.27], U(\lambda x.y), [] \rangle$
$\langle [], [y.27], [\lambda x.y\ U\ ap], [] \rangle$
$\langle [Aclosure], [y.27], [U\ ap], [] \rangle$
where Aclosure has fn $\lambda x.y$ and env $[y.27]$
$\langle [Bclosure\ Aclosure], [y.27], [ap], [] \rangle$
where Bclosure is the fn $\lambda f.\{\lambda y.f(y)\}(2)$ with env $[y.27]$
$\langle [], [f.Aclosure\ y.27], \{\lambda y.f(y)\}(2), D_1 \rangle$
where D_1 is $\langle [], [y.27], [], [] \rangle$
$\langle [2], [f.Aclosure\ y.27], [\lambda y.f(y)\ ap], D_0 \rangle$
$\langle [Cclosure\ 2], [f.Aclosure\ y.27], [ap], D_1 \rangle$
where Cclosure has fn $\lambda y.f(y)$ and env $[f.Aclosure\ y.27]$
$\langle [], [y.2f.Aclosure\ y.27], f(y), D_2 \rangle$
where D_2 is $\langle [], [f.Aclosure\ y.27], [], D_1 \rangle$
$\langle [2], [y.2f.Aclosure\ y.27], [f\ ap], D_2 \rangle$
$\langle [Aclosure\ 2], [y.2f.Aclosure\ y.27], [ap], D_2 \rangle$
$\langle [], [x.2y.27], y, D_3 \rangle$
where $D_3 = \langle [], [y.2f.Aclosure\ y.27], [], D_2 \rangle$
Note how the env binding y to 27 has been reinstated from Aclosure. Now we stack 27 and unwind...
$\langle [27], [x.2y.27], [], D_3 \rangle$
$\langle [27], [y.2f.Aclosure\ y.27], [], D_2 \rangle$
$\langle [27], [f.Aclosure\ y.27], [], D_1 \rangle$
$\langle [27], [y.27], [], [] \rangle$

We now show how the transform function of the SECD machine can be used to define a function mechanical.val which evaluates an AE x in an environment E. First we define a function iterate, which repeatedly applies a function f to a state S.

```
iterate (f, S) = null (sel-control (S)) →
                    (null (sel-dump (S)) → sel-stack (S),
                    iterate (f, reinstatedump (S)),
                    iterate (f, f(S))
                      where reinstatedump (S) = consontostack
                                (sel-stack (S), sel-dump (S))
```

Next we define a function load-AE (E, x) which loads an AE x in an environment E into an SECD state

$$\text{load-AE}(E, x) = \langle [], E, x, [] \rangle$$

Finally:

$$\text{mechanical.val}(E, x) = \text{iterate}(\text{transform}, \text{load-AE}(E, x))$$

Most writers on semantics (including Tennent) consider evaluate the 'real' meaning of AE's, and think of mechanical.val as a description of an implementation of AE's. The reasons for this is that the $\langle S, E, C, D \rangle$ model is closely related to runtime storage structures in recursive languages, and that mechanical.val specifies an evaluation order whereas evaluate doesn't. (this reflects again the inability of pure mathematical formalisms to incorporate the concept of process). Strictly speaking, mechanical.val and evaluate are different descriptions of the same function. The

description of mechanical val is inspired by, and suggests, a particular way to evaluate AE's ; but that does not alter its status as a function.

(Algol semantics - cf Brady p238ff).

NB Lucas (1971) points out that the state of a Landin; McCarthy or Vienna-style approach to semantics depends crucially on the language being defined, and he discussed a correlation between the complexity of the state needed to define a language using the Vienna machine and the complexity of the language.

7 Operation Semantics : VDL.

7.1 Intro (Brady)

One of the outstanding early achievements in formal semantics was the IBM Vienna Laboratories defn of the complex programming language PL/1. This project was based on Landin's AE defn of Algol 60 and McCarthy's abstract syntax / state transformation semantics. The overall aim of the Vienna definition project was to lay down precise and detailed conditions which had to be met by an implementation.

The over-riding concern for consistent implementation is reflected in the V lab's defn of 'meaning' as 'the entire sequence of states corresponding to a program run'. This has the advantage of being very precise, but is extremely difficult to reason with about the behaviour of a program.

Following McCarthy and Landin, the V defn interprets an abstact syntax representation of a program. One of the contributions of the Vienna work was a calculus of structured objects which is applicable equally to machine states and abstract syntax trees. The cornerstone of the calculus is a powerful replacement operator μ. As in the case of the SECD machine, one of the state components is a control, consisting of the piece of program currently being interpreted. Since the order in which expressions are evaluated is deliberately left partially unspecified, the control component is itself a tree, and is represented as a structured object, with instructions at the tree nodes. Any of the instructions at the leaves of the control tree is a candidate to be executed, so that there is generally a set of possible next states. Thus the function analogous to the SECD machine transform returns

a set of successor states, and a program execution is a sequence $S_1 S_2 \ldots$ of states such that $S_{i+1} \in \text{transform}(S_i)$. An execution step modifies the state and, in particular, the control tree. Two commonly occurring modifications to the control tree are:

- delete the current node and pass an appropriate value to higher nodes in the tree.
- replace the current node by a tree.

The VDL is unsatisfactory for reasoning about programs. Part of the problem seems to be that the sequence of states is completely unstructured. There is scope for modification, however, as the control tree really is a tree.

7.2 (Pagan) THE OPERATIONAL APPROACH VDL

We take as a starting point some suitable formulation of the abstract syntax of the subject language and assume that there will never be any attempt to use the semantic specifications for the analysis of syntactically invalid programs. In the operational approach, meaning is described in terms of such devices as abstract machines with discrete 'states' and more-or-less explicit sequences of computational operations. The operational approach is not directly oriented towards language implementation - taken to the extreme, an actual compiler or interpreter for a subject language can be regarded as the language's definitive specification. However, most compilers have features / restrictions which have little or nothing to do with the language definition. Thus most operational definitions specify an abstract rather than a real processor for the

subject language.
A compiler-oriented operational definition emphasizes the specification of a process for translating the subject language into some other (presumably simple) representation which is independent of the subject language. An interpreter-oriented def., on the other hand, is largely concerned with modelling the process of executing subject-language programs (usually in abstract form); the specifications for any necessary translations of programs from textual to abstract form are deemphasized or omitted. VDL is an interpreter-oriented method.
VDL is one of the oldest and best-known metalanguages for writing defns of formal systems that model the interpretation of (usually abstract) programs. The initial aim is to define an abstract machine for interpreting abstract programs of the subject language by passing through a sequence of discrete states, and to this extent its mode of operation is independent of the subject language. Actually, however, it is highly tailored to the subject language by virtue of the fact that the structure of its states and the allowable transitions from one state to another must be specifically defined. The allowable state transitions are defined by a set of instruction definitions written in a special notation; these typically make up the bulk of a VDL lang. spec.

7.3 Notation for Objects & Abstract Syntax

VDL incorporates a general notation for describing and manipulating hierarchical structures; these structures may be pictured as trees with labelled edges but no left-right ordering.

p136

An elementary object is an object with no internal structure of its own, so that it corresponds to a leaf node of a tree representing some structured object that contains it. A structured or composite object is denoted by a set of one or more pairs of the form $\langle s\ a \rangle$, where s is a selector and a denotes another object (either elementary or structured). A set of n selector-object pairs may be represented as a tree with n branches, each of which is labeled with one of the selectors and leads to a subtree representing the object selected. The presence of the selectors make left-right ordering unnecessary.

The null object Ω is a special elementary object which serves to indicate the "emptiness" of an object or absence of some component of an object; the null object (and its selector) have no corresponding branch in the tree diagram, so for example $[\langle s\text{-}p1\ A \rangle]$ is the same object as $[\langle s\text{-}p1\ A \rangle, \langle s\text{-}p2\ \Omega \rangle]$

A selector* may be used as a function that takes an object as an argument and yields the corresponding component of the object, considering the object as a tree, the function yields the subtree below the branch labeled by the selector. If the object does not have an object corresponding to the selector, the null object results. The selectors for the immediate components** of any object must be all different.

The mutation operator μ builds new objects which are modifications of old ones. An expression of the form $\mu(a; \langle s\ b \rangle)$ returns the object obtained by replacing the s component of a with the object b

There is often a need for objects which are to be regarded as lists of other objects, such that there may be any number

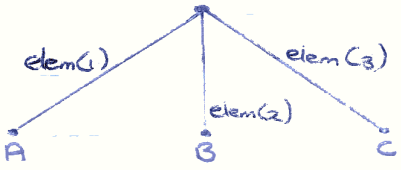
* by convention, selector names begin with 's-'

** immediate descendants in the tree.

of objects in a list. The convention is to use elem(1), elem(2) ... as the selectors of the list components (elements). The list

f = { <elem(1) : A>, <elem(2) : B>, <elem(3) : C> }

can be pictured as:



and can be abbreviated as <A, B, C>. elem(2)(f) = B, etc.

Standard functions:

- head ≡ elem(1) so head(f) = A
- tail as cdr in Lisp, so tail(f) = <B, C>
- length = # elts in list, so length(f) = 3

The concatenation operator for lists consists of a caret ^.

eg f ^ tail(f) = <A, B, C, B, C>

f ^ head(f) undefined, as head(f) is not a list.

f ^ <head(f)> = <A, B, C, A>

The empty list is <>

A predicate is a function p: object → {T, F}. By convention, names of predicates begin with 'is-'. Predicates which test for specific elementary objects are formed by appending the symbol for the object, eg is-'+'(opr), is-Ω(z). Predicates which test objects for membership in various classes must be explicitly defined by equations, eg

is-asmt-stmt = (<s-lhs : is-var>, <s-rhs : is-expr>)

states that an object will satisfy 'is-asmt-stmt' if it consists of a component satisfying is-var and selected by s-lhs, and a component satisfying is-expr and selected by s-rhs.

Predicates may also be defined as disjunctions of other predicates - eg

is-abcd = is-A ∨ is-B ∨ is-C ∨ is-D

By convention, predicates that test for lists end in '-list' and are not explicitly defined by equations, eg the predicate is-asmt-stmt-list is satisfied by any list of objects that all satisfy is-expr.

A VDL semantics specification includes a set of predicates describing the structure of a state Σ of the abstract machine. All VDL abstract machines have a control component in the state, a special object called the control tree

M.D. Both the program syntax and the state structure are described in the same syntactic notation. The advantages of the tree representation used are:

- closed under substitution of subtrees for subtrees
- unique access paths to components (cf DOAG's)
- finite access paths to components (cf DAG's)
- can represent DOAG's and DAG's by allowing composite selection operators at terminal vertices.

The VDL approach is based on abstract syntax (semantics of the program as a whole is defined by associating semantics with each syntactically-specified program component) and McCarthy's notion (1962) that "the meaning of a program is defined by its effect on the state-vector" (ie in terms of runtime transformations)

THE FORMALISM (NB slightly different notation to Pagan)

A Vienna Definition System V is a septuple $(E, C, \Omega, S, \sigma, \mu)$

where
- E is a set of elementary objects — $E \cap C = \Phi$
- C is a set of composite objects — $E \cup C = O$ (objects)
- Ω is the empty object — $\Omega \in E$
- S is a set of selectors — $S \subset E$
- σ is a general selector operator — $\sigma : S \times O \to O$
- μ is the construction (mutation) operator — $\mu : O \times S \times O \to C \cup \Omega$

Axioms:
1. $s(e) = \Omega \quad (s(e) \equiv \sigma(s, e))$ where $s \in S$, $e \in E$
2. $(e_1 = e_2)$ is decidable
3. $c_1 = c_2 \to \forall s,\ s(c_1) = s(c_2)$
4. $\sigma(s, \mu(o, s_1, o_1)) = (s = s_1 \to o_1,\ \sigma(s, o))$

Wegner: "To obtain ... understanding of a class of computational structures, it is necessary to consider both representation independent axiomatic characterizations and specific models that are realizations of the axiom system ..."

"$(E, C, \Omega, S, \sigma)$ with axioms 1-3 define a purely syntactic system ... for which equality of objects implies identity of structure. The operator μ with axiom 4 introduces semantic content ... in the sense that two different expressions containing ... μ may define the same ... object".

"Composite objects may be regarded as tree-structured computer memories. Composite selectors may be regarded as addresses that select locations in memory, where locations may have components that are locations* The object selected may be regarded as the value contained in the location"

* cf tree-structured filing systems.

The tree has selectors as edges, E\S objects as leaves, and C objects as branches.

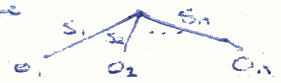
A programming language is characterised by a set of programs. Programs and program components are represented as Vanna trees. The composition of sets from simpler sets is represented by composition of predicates (cf V, Λ), eg:

$P = (\langle S_1 : P_1 \rangle, \langle S_2 : P_2 \rangle, \ldots, \langle S_n : P_n \rangle)$ defines the predicate satisfied by all objects that can be constructed from objects $x_1, x_2, \ldots, x_n$ satisfying the predicates $P_1, P_2, \ldots, P_n$ by the construction operator $\mu_o(\langle S_1 : x_1 \rangle, \langle S_2 : x_2 \rangle, \ldots, \langle S_n : x_n \rangle)$

where $\mu_o(\langle S_1 : O_1 \rangle, \langle S_2 : O_2 \rangle, \ldots, \langle S_n : O_n \rangle)$

$= \mu(\mu(\mu(\ldots(\Omega, S_1, O_1), S_2, O_2)\ldots), S_n, O_n)$

(which constructs tree



EXPRESSION EVALUATION

is-state = $(\langle S\text{-}C : is\text{-}control \rangle, \langle S\text{-}env : is\text{-}env \rangle, \langle S\text{-}O : is\text{-}output \rangle)$

The control tree has instructions at its nodes of the form

instructionname $(\bar{x})$ = { pred. → action } OR = a

$P_1 \rightarrow a_1$

$P_2 \rightarrow a_2$

$P_n \rightarrow a_n$

Instructions can be:

- macro (self replacing): syntactic transformations replacing the instruction with a subtree of instructions (expands tree, modifies only control)

- value-returning (PASS assignment): semantic transformations ↔ (strong correlation) delete instructions, pass values to predecessor vertices (reduces tree)

(Only) any terminal vertex can be executed at any time

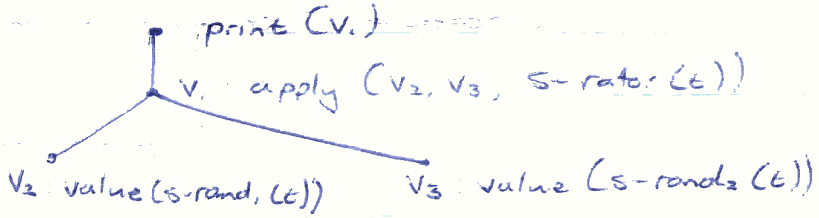
Example eval-expr (t) = print (a),
a: value (t)

(order ok as a first line a undefined until a has been evaled)

value (t) = is-binary (t) → apply (a, b, s-rator (t))
a: value (s-rand$_1$ (t))
b: value (s-rand$_2$ (t)),
is-var (t) → PASS ← t, s-env (ξ)
is-const (t) → PASS ← t

apply (val$_1$, val$_2$, op) = op = '+' → PASS ← val$_1$ + val$_2$
op = '*' → PASS ← val$_1$ * val$_2$

CONTROL TREE



v$_1$, v$_2$ and v$_3$ are used as a, b and t are bound variables and hence will be different actual variables at different parts of the tree

Block-Structure Environments

We add denotation (store) and attribute tree to the state, giving:

is-state = (⟨s-c : is-control⟩, ⟨s-env : is-env⟩ (ids → cells), ⟨s-den : is-den⟩ (cells → values),
⟨s-at : is-attribute⟩, ⟨s-d : is-dump⟩, ⟨s-n : is-integer⟩)

is-environment = ({⟨id : is-number⟩ || is-id(id)} such that)

is-attribute = ({⟨n : is-type⟩ || is-n(n)})

is-denotation = ({⟨n : is-value V is-procden V is-funcden⟩ || is-n(n)})

is-procden = (⟨s-attr : is-procattr⟩, ⟨s-env : is-env⟩)

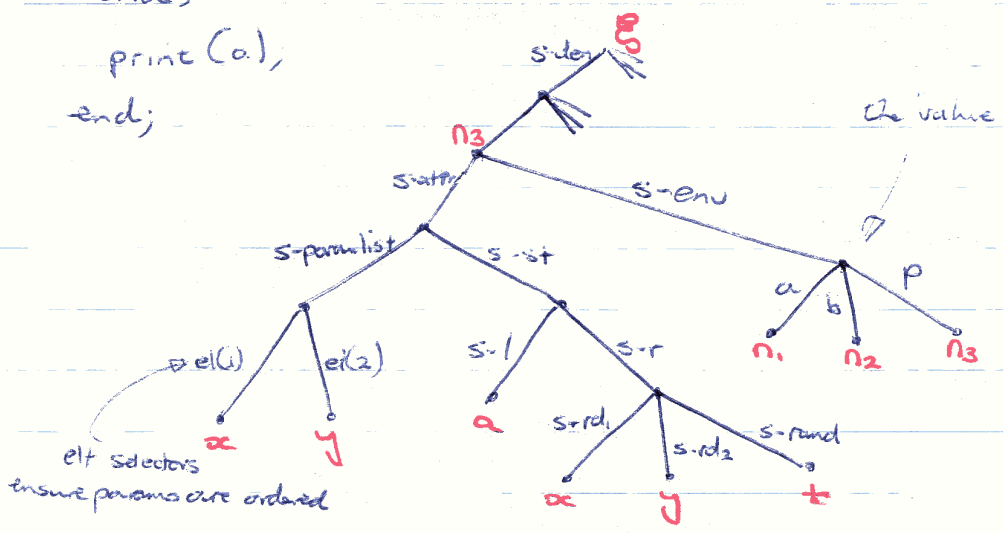
is-procattr = (⟨s-paramlist : is-id-list⟩, ⟨s-st : is-st⟩)

(the value of a procedure is its carried environment, formal parameters, and body statements)

The initial state is (interpret(t), ⟨⟩, ⟨⟩, ⟨⟩, ⟨⟩, 1)

Example 1

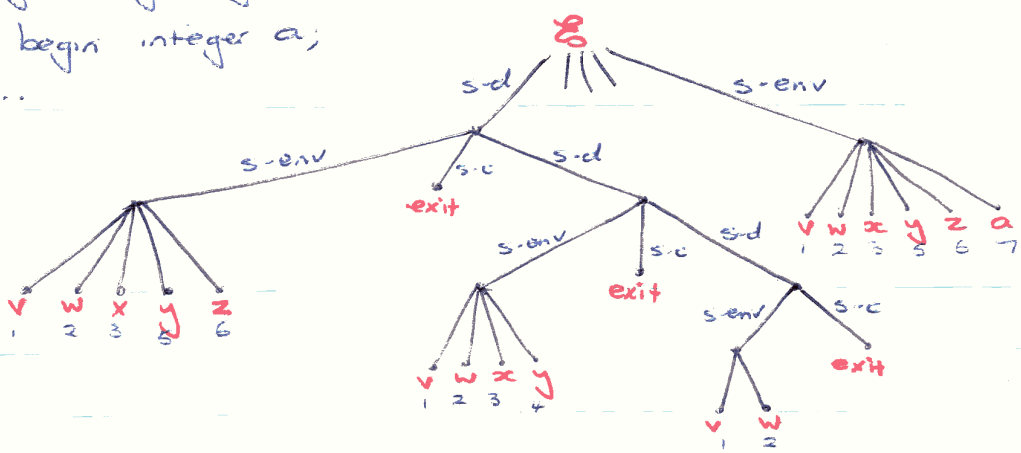
```
begin integer a, b;                    (a: n1, b: n2)
    procedure P(x, y); a := x + y;     (P: n3)
    begin integer a;
        a := 1, P(a, a), print(a);
    end;
    print(a),
end;
```



? $is\text{-}d = (\langle s\text{-}env : is\text{-}env \rangle \langle s\text{-}c : is\text{-}control \rangle \langle s\text{-}d : is\text{-}dump \rangle) \vee is\text{-}\Omega$

Ex2

B1 : begin integer v, w;
B2 : begin integer x, y;
B3 : begin integer y, z;
B4 : begin integer a;
...



SYNTHESIS (BRIEF NOTES)

8. CODE GENERATION

INTERMEDIATE FORMS OF TRANSLATION

- Size and alignment of elementary storage objects
  eg placement of numbers used to build composite objects
- Composition of allocatable storage blocks
  eg heap objects, activation records at compile time
  (eg packed structures smaller than unpacked, etc).
- Register allocation 1 : bookkeeping for blocks, keeping some values in fast registers.
- Register allocation 2 : temporaries for expressions
- Code reduction
  - stores - reducing need for temporaries (eval order)
  - operators - try to simplify expressions
  - load - check if something which must be loaded is already loaded

We now have a computational graph (see page ①)

- REGISTER-ASSIGNMENT ALGORITHM (simulation of eval order)
  Registers have the following attributes: free/copy as such/unique copy (ie if we need this register we must save its contents first)/locked (in the process of placing a value) A table of registers is kept.
- INSTRUCTION SEQUENCE SELECTION (decision table - leaf conditions and root tables make action sequences)
  This produces a target tree (cf assembler)
- Internal address resolution - access paths (ie change labels to addresses)
- External address resolution (link-load)

9. Error Handling.

9.1 What the Compiler should tell the User.

- negative feedback (errors) for debugging
- positive feedback (listings) for maintenance
- reporting should be as comprehensive as possible without interfering with the listings excessively
- positive feedback:
  - source & target language, compiler name & version, date & time of compilation, options on & off
  - text, co-ordinate system (line, statement, location #, nesting), cost measures (statistics)
  - pretty-printing, target code options
  - cross-reference & symbol-table printout options.

9.2 Error Reactions.

- The compiler can react with:

	when
Invalid translation	no redundancy detected
Crash or loop	error sensitivity occurs
Quit	error detection & report used.
Recover & continue reporting	error recovery used
Repair & continue compiling	error repair used

NB Some errors transform valid programs into other valid programs, and only manifest themselves dynamically. LL(1) and LR parsing detects syntax errors at the earliest point possible before accepting the first symbol to offend against validity.

Recovery Techniques

We can change the stack (unpleasant) or change the control, either by inserting tokens or skipping tokens. Insertion is difficult. The most widely used method is:

Panic mode: discard control until something "solid" is found (eg statement delimiter); then discard stack until something that can validly precede (succeed) the solid token is found. We build up a pack of solid symbols – the price a correct program must pay for error recovery.

Alternatives

- error isolation (smallest reducible phrase containing error)
- error productions — $A \to \underline{\text{error}}$ ad hoc additions to grammar to deal with "most common" errors
- combination of these

Wirth & Good: Find a continuation, discard to an anchor, and replace with something consistent

$$w t X \in (T^* \setminus L) \to w t' X' \in L$$

stack; error token; control; all token combinations; valid programs

Find $w\mu \in L$; construct the set of anchors
$\{ d \in T : w\nu\alpha\ldots \in L \wedge \nu\ldots = \mu \}$
Find the shortest $\eta \in T^*$ such that $t X = \eta\alpha\mu'$;
Replace η with the shortest $\nu \in T^*$ such that $w\nu\alpha\ldots \in L$.

NB: Errors are arbitrary & irrational – we cannot build a rational system to deal with them systematically.

9.4 Some Remarks on Error Types.

Levels

• Notes	eg reporting of non-standard constructs
• Comments	programming style
• Warnings	possible error.
• Errors	reparable
• Fatal errors	recoverable
• Deadly errors	Goodbye!

There should be an option to set the report level, with a cut-off on density (avalanche)

ANALYSIS - lexical, syntactic, semantic ↓

NB Recovery based upon redundancy which increases in direction of arrow (so easiest to deal with semantic errors)

SYNTHESIS - detects only "compiler errors", including violation of table limits, etc

RUNTIME - The compiler can prepare for these by some backmapping from store to source data declarations and/or some backmapping from target code to source text.