In the following α, β, γ, and δ are strings of zero or more terminals and/or <nonterminals>

Left Recursion

$A \rightarrow A\ \alpha_1 \mid \ldots \mid A\ \alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ where $\beta_1 \ldots \beta_n$ do not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta_1\ A' \mid \ldots \mid \beta_n\ A'$

$A' \rightarrow \alpha_1\ A' \mid \ldots \mid \alpha_m\ A' \mid \varepsilon$ an equivalent grammar

Left Factorization

A -> $\alpha\beta$ | $\alpha\gamma$ | δ

Rewrite by combining the common prefix:

A -> αA' | δ

A' -> β | γ

Nullable

$$
\begin{array}{lcl}
Nullable(\varepsilon) & = & true \\
Nullable(\mathtt{a}) & = & false \\
Nullable(\alpha\beta) & = & Nullable(\alpha) \wedge Nullable(\beta) \\
Nullable(N) & = & Nullable(\alpha_1) \vee \ldots \vee Nullable(\alpha_n), \\
 & & \text{where the productions for } N \text{ are} \\
 & & N \rightarrow \alpha_1, \quad \ldots \quad , N \rightarrow \alpha_n
\end{array}
$$

where a *is a terminal,* N *is a nonterminal,* α *and* β *are sequences of grammar symbols and* ε *represents the empty sequence of grammar symbols.*

First Sets

$$
\begin{array}{lcl}
FIRST(\varepsilon) & = & \emptyset \\
FIRST(\mathtt{a}) & = & \{\mathtt{a}\} \\
FIRST(\alpha\beta) & = & \begin{cases} FIRST(\alpha) \cup FIRST(\beta) & \text{if } Nullable(\alpha) \\ FIRST(\alpha) & \text{if not } Nullable(\alpha) \end{cases} \\
FIRST(N) & = & FIRST(\alpha_1) \cup \ldots \cup FIRST(\alpha_n) \\
 & & \text{where the productions for } N \text{ are} \\
 & & N \rightarrow \alpha_1, \quad \ldots \quad , N \rightarrow \alpha_n
\end{array}
$$

Follow Sets

FOLLOW(A) algorithm:

If A=S then put end marker (*e.g.*, EOFtk) into FOLLOW(A) and continue

Find all productions with A on rhs: Q -> αAβ

- if β begins with a terminal q then q is in FOLLOW(A)
- if β begins with a nonterminal then FOLLOW(A) includes FIRST(β) - {ϵ}
- if β=ϵ or when β is nullable then include FOLLOW(Q) in FOLLOW(A)

Course grammar:

<S> -> **program** <V> <B>
<B> -> **begin** <V> <Q> **end**
<V> -> empty | **var identifier .** <V>
<M> -> <H> **+** <M> | <H> **-** <M> | <H> **/** <M> | <H> ***** <M> | <H>
<H> -> **&** <R> | <R>
<R> -> **identifier** | **number**
<Q> -> <T> **#** <Q> | empty
<T> -> <A> **,** | <W> **,** | <B> | <I> **,** | <G> **,** | <E> **,**
<A> -> **scan identifier** | **scan number**
<W> -> **write** <M>
<I> -> **if [** <M> <Z> <M> **]** <T>
<G> -> **repeat [** <M> <Z> <M> **]** <T>
<E> -> **let identifier :** <M>
<Z> -> **<** | **>** | **:** | **=** | **==**

Grammar semantics:

- Delimiters:
  - . , # [ ]
- Less than and greater than are standard symbols
  - < >
- '&' means absolute value for number
  - Number may be immediate or stored at location indicated by identifier
- The following operators have standard definitions, except no precedence
  - + - * /
  - Operations are applied from left to right
- 'let identifier : M' assigns the value of M to the given identifier
- '=' and '==' are both treated like standard '=='
  - Assignment of values to variables can only be performed using '[let] identifier : M'
- 'if [M Z M] T'
  - Means to do T if and only if (M Z M) is true (always true if Z is ':')
- 'repeat [M Z M] T'
  - Means to repeat T until (M Z M) is false
- When scanning an identifier, allocate memory using the identifier as the variable name and set value to zero
- When scanning a number, allocate memory using a temporary variable name and set value to the number

Assembly Language:

- BR (1, jump to arg)
- BRNEG (1, jump to arg if ACC <0)
- BRZNEG (1, jump to arg if ACC <=0)
- BRPOS (1, jump to arg if ACC >0)
- BRZPOS (1, jump to arg if ACC >=0)
- BRZERO (1, jump to arg if ACC ==0)
- COPY (2, arg1 = arg2)
- ADD (1, ACC = ACC +arg
- SUB (1, ACC = ACC - arg)
- DIV (1, ACC = ACC / arg)
- MULT (1, ACC = ACC * arg)
- READ (1, arg=input integer)
- WRITE (1, put arg to output as integer)
- STOP (0, stop program)
- STORE (1, arg = ACC)
- LOAD (1, ACC=arg)
- NOOP (0, nothing)

Virtual Machine Architecture: