# PLO - UN SEMPLICE LINGUAGGIO REALE

- **Ma come è fatto un linguaggio reale?**
- **Come sono fatte le categorie sintattiche nella loro forma più semplice**

- Linguaggio Pascal-like
- Singolo tipo di dato INTEGER (Booleani rappresentati come interi)
- Strutture di controllo standard
- Astrazione del controllo (Procedure senza parametri)

```
<program>   P → B.
<block>     B → D S
<declar>    D → [const I=N{,I=N};]
              | [var I{,I};]
              | [procedure I;B;]
<stmts>     S → ε | I := E
              | call I
              | if C then S
              | while C do S
              | begin S{;S} end
<Cond>      C → odd E | E > E
              | E >= E | E = E
              | E # E | E < E
              | E <= E
<Expr>      E → I | N | E bop E
              | ( E )
```

# SINTASSI DI PLO

- I programmi P sono blocchi.
- I blocchi B sono una dichiarazione D seguita da un comando S. Si noti che D ed S in modo libero dal contesto, significa che quello che possiamo generare da S non deve avere necessariamente nessun vincolo con quello che generiamo da D, in altre parole questa grammatica non ci obbliga in nessun modo ad usare identificatori precedentemente definiti. Quindi dalla descrizione sintattica rimangono fuori aspetto context sensitive che la grammatica CF non può rappresentare, servirà una definizione di una "semantica statica" (che abbiamo chiamato analisi semantica) che verificherà questi vincoli prima di compilare o eseguire.
- Le dichiarazioni D sono dichiarazioni di valori costanti con nome (const), dichiarazioni di variabili (var) e dichiarazioni di procedure (ovvero di un blocco riferibile con un nome).
- I nomi sono gli identificatori I considerati qui parte del vocabolario; N sono i numeri naturali, considerati parte del vocabolario.
- Non abbiamo bisogno del concetto di tipo in quanto abbiamo solo un tipo di dato.
- Si noti che le produzioni di D sono tutte opzionali, questo significa significa che un programma può essere un blocco con nessuna dichiarazione.
- I comandi sono: il comando vuoto, l'assegnamento di un valore (denotato da una espressione) ad un identificatore, la chiamata ad una procedura mediante il suo nome, il comando condizionale che esegue un comando al verificarsi di una condizione booleana C, il ciclo che ripete un comando finché una data condizione C è vera e la composizione sequenziale di comandi.

- Le condizioni C sono espressioni dal valore booleano: Odd è un predicato unario che verifica se l'espressione è 0, poi ci sono vari operatori di confronto di espressioni E.
- Le espressioni E sono identificatori, valori naturali, operazioni tra espressioni e espressioni tra parentesi.
- Qui sintetizziamo con op la metavariabile dell'insieme {+,-,*,/}. La stessa cosa potevamo farla per gli operatori di confronto, e mettere in C la produzione in E cop E con cop metavariabile in { >,>=,=,#,<=,< }.

Per la grammatica delle espressioni dobbiamo fare attenzione all'ambiguità: questa è ambigua non considerando le parentesi. Se invece mettiamo tutte le parentesi, allora non abbiamo più l'ambiguità ma diventa un linguaggio pesante da usare, prono ad errori (tante parentesi da scrivere) e difficile da leggere. Dobbiamo quindi cercare un compromesso che permetta di tener conto le precedenze matematiche in assenza di parentesi, senza obbligare ad usare le parentesi e comunque eliminando l'ambiguità.

```
<Expr>      E → [+,-] T [A T]
<Add op>    A → + | -
<Terms>     T → F [M F]
            M → * | /
<Factor>    F → I | N | ( E )

<Numbers>   N → [+,-] d{d}
<Digit>     d → 0 | … | 9

<Ident>     I → l{l,d}
<Letter>    l → A | … | Z
```

Questa grammatica delle espressioni rispetta tutte le convenzioni e non è ambigua. Questa è la definizione che serve ai compilatori, mentre per parlare di semantica possiamo anche semplicemente usare la grammatica ambigua, più facile da usare.

Per il compilatore, la grammatica deve essere completa, quindi dobbiamo specificare tutti gli elementi. L'identificatore è una stringa che deve necessariamente iniziare con un carattere non numerico. Qui ci sono solo i caratteri maiuscoli, ma è molto facile estendere. È da notare che questo ultimo gruppo di grammatiche (N e I) non sono parte del parsing ma dell'analisi lessicale (scanning) per questo non è tipicamente considerata parte della grammatica principale. Ricordiamo che lo scanning prende il programma e lo divide in parole/lessemi che descrivono entità del programma (parole, identificatori, costanti..) e poi lo si guarda come un singolo elemento. Questo è usato anche per raccogliere le informazioni sui tipi, la fase di scan quindi fa l'operazione di convertire una sequenza di caratteri in una sequenza di lessemi, ovvero di elementi corredati dal "tipo". Questo significa che, se ci mettiamo al livello della grammatica dei comandi, allora possiamo trattare semplicemente identificatori e numeri come insiemi noti N = INTEGERS, I = {id, rate, ciao2,…}. Questo è un esempio di come la potenza delle CFG permetta di modificare la grammatica, ottenendo nuove grammatiche equivalenti ma più adatte magari al processo di elaborazione per il quale vengono definite (compilazione). Quindi le CFG permettono di descrivere i linguaggi ad un profondo livello di dettaglio che non è necessario conoscere per utilizzare il linguaggio. Solitamente l'utente deve conoscere la sintassi astratta del linguaggio. Quindi i punti di vista dell'implementatore del dell'utente sono diversi: è necessaria una grammatica abbastanza grossolana magari senza ambiguità per l'utente che è interessato solo all'abstract sintax tree, dove serve sapere se scrivendo in un modo o in un altro ci sono differenze **semantiche**, ma si necessita una grammatica abbastanza fine per evitare problemi di ambiguità ad esempio nell'implementazione di un parser.