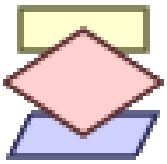

Manuale Flowgorithm
Paradigma di Programmazione
- I modelli -

Abbinato alla Versione 2.15 di Flowgorithm

Roberto Atzori

« L'informatica non riguarda i computer più di quanto l'astronomia riguardi i telescopi. »

(Edsger Wybe Dijkstra)

Sommario

Paradigma di Programmazione

Introduzione

In informatica, un **paradigma di programmazione** è uno stile fondamentale di programmazione, ovvero un insieme di strumenti concettuali forniti da un linguaggio di programmazione per la stesura del codice sorgente di un programma, definendo dunque il modo in cui il programmatore concepisce e percepisce il programma stesso. Diversi paradigmi si differenziano per i concetti e le astrazioni usate per rappresentare gli elementi di un programma (come ad esempio le funzioni, gli oggetti, le variabili, vincoli, ecc.) e per i procedimenti usati per l'esecuzione delle procedure di elaborazione dei dati (assegnazione, calcolo, iterazione, data flow, ecc).

Tipi di paradigmi

Per quanto riguarda almeno i linguaggi di programmazione usati nella pratica industriale dello sviluppo del software, si può identificare un "filone principale" di paradigmi di programmazione:

programmazione modulare

(metà anni settanta) Modula, CLU (linguaggio)

programmazione orientata agli aspetti

Una estensione dell'OOP (anni 2000) AspectJ

programmazione orientata agli utenti

Inside Microsoft Windows NT Internet Development (1998 Microsoft Press), piattaforma .NET

programmazione orientata agli oggetti

(anni ottanta) Smalltalk, Eiffel, C++, Java, Python, Ruby, piattaforma .NET

programmazione strutturata secondo patterns

Java blueprints, Sun Java blueprints

programmazione per pattern matching

(Espressioni regolari)

programmazione procedurale

(anni sessanta) Fortran, F#

programmazione strutturata

(primi anni settanta) Pascal, C

programmazione per tipi di dati astratti

(tardi anni settanta) OBJ

Altri paradigmi sono nati per applicazioni specifiche:

programmazione concorrente

Inizialmente per il calcolo su architetture parallele (anni settanta) Erlang, Communicating Sequential Processes (CSP), Occam

programmazione logica

Per applicazioni euristiche, intelligenza artificiale, ecc.; (anni settanta) Prolog

programmazione funzionale

Per applicazioni matematiche e scientifiche ecc.; (anni settanta) Lisp, Haskell

programmazione orientata agli eventi

Per applicazioni real-time e interfacce grafiche

programmazione a vincoli

Flowgorithm

I meccanismi di astrazione dei linguaggi di programmazione, differenti per i vari paradigmi, possono contribuire a rendere possibile stili di programmazione basati su concetti non direttamente forniti dal linguaggio utilizzato.

Con l'uscita della versione 2.15.0, Flowgorithm si arricchisce di una nuova funzionalità: il supporto per la personalizzazione dei modelli di programmazione.

Il Visualizzatore Codice Sorgente di Flowgorithm consente di convertire i diagrammi di flusso in diversi linguaggi di programmazione attualmente in voga. Questi includono: Java, C #, Visual Basic .NET, Smalltalk e molti altri. Il codice sorgente generato viene creato utilizzando i modelli di programmazione. Si tratta di file che contengono informazioni sintattiche sul linguaggio di programmazione selezionato come parole chiave, formattazione e precedenza degli operatori.

Questa documentazione individua il formato e i modelli utilizzati da Flowgorithm. Tutto ciò può essere utile per creare nuovi modelli partendo da quelli esistenti.

Estensione del file

I modelli di programmazione di Flowgorithm usano l'estensione .**fpgt** anche se il Visualizzatore del Codice Sorgente è in grado di aprire file con qualsiasi estensione purché rappresenti un modello di programmazione.

Formato File

I modelli di programmazione vengono salvati in un semplice file di testo utilizzando il formato **INI**. L'obiettivo è renderli di facile lettura e scriverli tramite l'uso di un semplice editor di testo. I modelli di programmazione di Flowgorithm supportano la codifica Unicode con la possibilità, quindi, di utilizzo di caratteri diversi da quelli proposti dalla codifica standard ASCII.

I files con estensione INI riconoscono tre elementi sintattici di base: commenti, intestazioni di sezione, e dichiarazioni di chiave o di valore. Partendo da questi, si è in grado di rappresentare tabelle non gerarchiche.

Gli elementi sono i seguenti:

Elemento	**Azione**	**Esempio**
Comment	I Commenti non producono alcuna azione ma sono utili come documentazione per chiunque legga il file.	; Comment
Section	Le Sezioni individuano differenti parti logiche del file.	[Section Name]
Key/Value	Le Chiavi (Key) si comportano come le variabili e vengono valorizzate.	Key = Value

Il nome e la chiave della sezione vengono utilizzati per identificare univocamente un valore. Questo è, più o meno, equivalente alle tabelle e alle righe presenti negli archivi di dati (databases).

Esempio
[Variable ID] Convention = camel Normal = {Name} Conflict = var_{Name}

Formato della linea

Ogni riga nel modello contiene fino a tre sotto valori. Ciò consente di specificare quanto segue per ogni riga:

1. Formato testo / valore
2. Flag che indica quando la linea è considerata valida
3. Qualsiasi cambiamento nell'indentazione

Formato linea
Text | Flags | Indentation

I tre valori non devono necessariamente essere presenti tutti nella riga. Se l'indentazione è lasciata vuota, ad esempio, il sistema assumerà il valore 0 (zero) o il significato di nessuna modifica.

Campi

Molte delle diverse chiavi / valori contengono "campi" che sono indicati da parentesi graffe '**{**' (parentesi graffa aperta) e '**}**' (parentesi graffa chiusa). Quando viene generato un programma, il testo viene inserito in questi campi a seconda dell'oggetto corrente (ad es. una espressione, una istruzione, ecc.). I contenuti dei campi possono essere prelevati direttamente dall'oggetto o, in alcuni casi, generati utilizzando altre parti del modello.

I nomi dei campi non sono 'case sensitive' (ovvero non c'è alcuna distinzione fra maiuscole e minuscole). L'utilizzo di una doppia parentesi graffa aperta '**{{'** indica una sequenza di sostituzione.

Nell'esempio seguente, la chiave '**Text'** contiene due campi: **{Variable}** e **{Expression}**.

Esempio – Istruzione di Assegnazione
Text = Set {Variable} = {Expression}

La maggior parte dei campi sono specifici per la sezione nella quale vengono utilizzati. Tuttavia, ci sono alcuni campi definiti 'globali' molto utili. Sono progettati per superare i limiti del formato del modello di programmazione.

Campi Globali	Contenuto
{space}	Questo contiene un singolo carattere (spazio). Poiché il formato del file elimina automaticamente spazi iniziali e/o finali, questa variabile può essere utilizzata per aggiungere esplicitamente un carattere spazio all'interno di un programma.
{pipe}	Il simbolo pipe '|' è usato per separare le tre sezioni di una linea. Per consentire l'uso del simbolo pipe nel programma, questa variabile contiene un singolo carattere '|'.

Flag

Quando viene generato il codice sorgente, il sistema può includere o escludere le righe in base alle informazioni dello stesso programma. Ciò può essere dovuto al fatto che sia necessaria una specifica sintassi, in presenza di diverse chiamate di libreria (in base al tipo di dati utilizzato in una istruzione), ecc ....

Per controllare quali linee sono valide nel modello di programmazione, il sistema fa uso dei Flags. I Flags sono definiti dalla sezione nella quale hanno un significato e possono essere utilizzati per controllare il codice sorgente generato. Si possono utilizzare più Flags ma separati da virgole. La linea è considerata valida se, e solo se, tutti i Flags sono abbinati in maniera corretta. Se un Flag è preceduto dal simbolo tilde '**~**', il sistema lo individuerà come essere '**False**'.

L'esempio seguente utilizza diversi Flags, in particolare "inc" e "step". Nel contesto di una struttura iterativa For, il Flag "Step" è impostato su True nella struttura iterativa For avente un valore di step diverso da 1 e "inc" è impostato a True se la struttura iterativa For segue un andamento in ordine crescente (positivo).

La combinazione di questi valori comporterà la selezione di una riga (che inizia con '**for**').

Esempio – Struttura iterativa "For"

```
[For]
Text = for {Variable} = {Start} To {End}                  | inc, ~step
     = for {Variable} = {Start} To {End} Step {step}      | inc, step
     = for {Variable} = {Start} To {End} Step -1          | ~inc, ~step
     = for {Variable} = {Start} To {End} Step -{step}     | ~inc, step
     = -->BLOCK                                           |              | 1
     = end for
```

Indentazione

La maggior parte dei linguaggi di programmazione consente l'uso dell'indentazione per formattare visivamente il testo. Ciò consente al programmatore di poter facilmente distinguere e comprendere blocchi e ad altri elementi sintattici. L'ultimo sotto valore di una riga può contenere la modifica nella indentazione del codice generato.

Nell'esempio seguente, la riga che contiene il valore speciale "**---> BLOCK**" (che ha un flag vuoto) ha un singolo "1" per l'indentazione. Ciò farà indentare il testo inserito di 1 livello.

Nota: questo **non** si riferisce ai caratteri, ma al livello di indentazione. Il livello di indentazione, nel codice generato, viene convertito automaticamente in un numero fisso di spazi.

Esempio – Struttura iterativa "While"

```
Text = while {condition}
     = -->BLOCK          | | 1
     = end while
```

Flags Globali

I modelli contengono un numero di flag a seconda se vengono utilizzate determinate istruzioni e funzioni. Questi sono globali e sono progettati per essere utilizzati con la sezione **[Program]** e la sezione delle istruzioni.

Flags	**Da utilizzare quando ….**
arccos	la funzione ArcCos è utilizzata nel programma.
arcsin	la funzione ArcSin è utilizzata nel programma.
arrays	vengono utilizzati gli Arrays nel programma.
char	la funzione ArcSin è utilizzata nel programma.
input	viene utilizzato il comando Input nel programma.
log10	la funzione Log10 è utilizzata nel programma.
output	viene utilizzato il comando Output nel programma.
pi	viene utilizzata la costante PI nel programma.
random	la funzione Random è utilizzata nel programma.
size	la funzione Size è utilizzata nel programma.
sgn	la funzione Sgn è utilizzata nel programma.
tofixed	la funzione ToFixed è utilizzata nel programma.
tostring	la funzione ToString è utilizzata nel programma.

Programma

Quando un diagramma di flusso (flowchart) viene convertito in un determinato linguaggio di programmazione, il sistema parte dalla sezione **[Program]**. Questa sezione dà, allo sviluppatore, la possibilità di inserire la funzione principale, le funzioni aggiuntive, le intestazioni di funzione e altre definizioni necessarie. Come conseguenza, nella sezione si farà un uso importante dei flags.

Esempio Java

```
[Program]
Text = import java.util.*;
     = import java.lang.Math;
     =
     = public class JavaApplication {{
     = private static Scanner input = new Scanner(System.in); | input   | 1
     = private static Random random = new Random();           | random  | 1
     =
     = -->MAIN                                                |         | 1
     = -->FUNCTIONS                                           |         | 1
     =                                                        | tofixed | 1
     = private static String toFixed(double value, int digits) {{ | tofixed | 1
     = return String.format("%." + digits + "f", value);      | tofixed | 2
     = }                                                      | tofixed | 1
     = }
```

Nell'esempio precedente i flags sono utilizzati, opzionalmente, per creare dichiarazioni per la funzione **Scanner** e **Random** rispettivamente. Inoltre, se viene utilizzata la funzione **ToFixed**, il modello creerà una funzione locale chiamata **ToFixed()** che implementerà tale logica.

La chiave testuale

La chiave testuale viene utilizzata per generare la sintassi del programma principale. Essa non contiene alcun campo.

Campi
Nessuno

Flags
Vedere l'argomento **Flags Globali**

Per poter inserire il blocco della struttura iterativa **While**, si utilizzi una singola linea contenente i seguenti ***valori speciali***. Se si vuole cambiare l'indentazione del blocco, assicurarsi di specificare l'indentazione dopo il secondo simbolo di pipe '**|**'.

Valore speciale	**Cosa fa**
-->HEADERS	Inserisce intestazioni di funzione. Queste verranno generate all'interno della sezione **[Function Header]**
-->MAIN	Inserisce la funzione principale. Questa è definita nella sezione **[Main Function]**
-->FUNCTIONS	Inserisce le funzioni dei programmi. Non viene incluso il **Main**. Queste sono definite nella sezione **[Function]**

Identificatori e Tipi

Attributi del linguaggio

La prima sezione del modello definisce un importante numero di attributi del linguaggio di programmazione di destinazione quali il suo nome, l'elenco delle parole chiave e se il linguaggio è '**case sensitive**'

Esempio Java
[Language] Name = Java Extension = java Direction = left-to-right Keywords = abstract, assert, boolean, break, byte, case, catch, char, class, const = continue, default, double, do, else, enum, extends, false, final, finally = float, for, goto, if, implements, import, instanceof, int, interface, long = native, new, null, package, private, protected, public, return, short, static = strictfp, super, switch, synchronized, this, throw, throws, transient, true, try = void, volatile, while Conflicts = input Case Sensitive = true Options = aligned

Attributi

Attributi	**Contenuti**
Name	Il nome del linguaggio di programmazione. Questo valore verrà visualizzato nel menù e negli altri elementi grafici dell'interfaccia utente (**GUI**)
Extension	Quando viene salvato un programma generato, questo valore rappresenta l'estensione del file
Direction	Questo attributo controlla se il codice generato verrà visualizza nel formato da sinistra verso destra (la tipica modalità di scrittura utilizzata nelle nazioni come Inghilterra, Spagna, Italia,..) o da destra verso sinistra (la tipica modalità di scrittura tipica di alcune nazioni quali Arabia Saudita, Israele, …)
Keywords	Questa chiave è utilizzata per elencare tutte le chiavi del linguaggio di programmazione di destinazione. L'elenco verrà utilizzato per generare **identificatori** univoci per il resto del programma. E' importante elencare tutte le **parole chiave** e le **parole riservate** che potrebbero causare errori sintattici quando vengono usati dall'utente come un qualsiasi identificatore.
Conflicts	Questa chiave viene utilizzata con le stesse modalità della chiave precedente **Keywords**. Essa fornisce un elenco delle **parole chiave** o **parole riservate** che verrà utilizzato per la generazione di identificatori. Comunque, questo elenco viene predisposto per elencare gli identificatori che lo stesso modello crea. Ad esempio, se un modello crea una funzione chiamata **Output**, allora la parola 'output' dovrebbe essere elencata qui.

Case Sensitive	Può essere impostata sia a **True** che a **False**
Options	Campi di formattazione opzionali supportati dal Visualizzatore del Codice Sorgente. Attualmente è supportato solo il valore '**aligned**' che crea un elenco a discesa contenente le parole chiave '**aligned**' e '**hanging**' per l'uso nei linguaggi di programmazione della famiglia del linguaggio di programmazione '**C**'

Identificatori

La maggior parte di linguaggi di programmazione possiede un elenco di parole chiave (che seguono lo stesso formato degli identificatori) che implementano i diversi elementi sintattici del linguaggio di programmazione. Ad esempio, la maggior parte dei linguaggi di programmazione utilizza la parola chiave "if" per le istruzioni condizionali.

È possibile creare un nome di variabile (o un nome di funzione) in un linguaggio che potrebbe non essere valida per un altro linguaggio. I programmatori di Visual Basic, ad esempio, possono creare una variabile chiamata "float" poiché non è una parola chiave (VB utilizza "single" per numeri in virgola mobile a precisione singola). Sebbene sia valido in VB, l'identificatore non può essere utilizzato testualmente se il programma viene convertito in Java. In questo caso, "float" è una parola riservata. Il conflitto risultante causerà un codice generato non valido (generando ad esempio un errore di sintassi).

Creazione di nuovi identificatori

I modelli contengono due sezioni progettate per generare nuovi identificatori (se necessario). Entrambe le sezioni contengono le stesse chiavi e la stessa modalità di base. L'identificatore originale di Flowgorithm viene controllato, ogni volta, con l'elenco di valori presenti nelle sezioni **Keywords** e **Conflicts** definiti nella sezione **[Language]**. Se non viene trovato alcun conflitto, verrà utilizzata la chiave **Normal** per creare l'identificatore. Se viene rilevato un conflitto, verrà utilizzata la chiave **Conflict**.

Esempio Java

```
[Function ID]
Convention = camel
Normal     = {Name}
Conflict   = func_{Name}

[Variable ID]
Convention = camel
Normal     = {Name}
Conflict   = var_{Name}
```

Nota: il carattere di sottolineatura '_' non è consentito negli identificatori di Flowgorithm. Può essere usato per creare una stringa univoca prefissa (o suffisso) di caratteri.

La chiave 'Convention'

La chiave **'Convention'** viene utilizzata per creare un identificatore utilizzando la convenzione di denominazione della lingua di destinazione. Può contenere una delle due opzioni:

1. Proper
2. Camel

L'opzione **Proper** viene utilizzata da linguaggi come Visual Basic e Smalltalk. In questa convenzione, la prima lettera di ogni parola è maiuscola come TestProgram e FirstName. Le lingue, come Java e C#, usano questa convenzione solo per i nomi di classe.

L'opzione **Camel** viene utilizzata per nomi di variabili in lingue come C # e Java. In questa convenzione, la prima lettera della parola è in minuscolo mentre il resto è in maiuscolo o minuscolo. Ad esempio: testProgram e firstName.

Campi
Nessuno

Flags
Nessuno

La chiave 'Normal'

Viene utilizzata questa chiave se l'identificatore, utilizzato nel diagramma di flusso, non è in conflitto con le parole chiave del linguaggio di destinazione.

Campi	
{Name}	Identificatore originale di Flowgorithm

Flags	**Da utilizzare quando ….**
integer	L'identificatore è di tipo Integer.
real	L'identificatore è di tipo Real.
boolean	L'identificatore è di tipo Boolean.
string	L'identificatore è di tipo String.
None	L'identificatore non è di alcun tipo conosciuto. Ciò accade quando, ad esempio, si è in presenza di un identificatore non dichiarato. Suggerito come impostazione di sintassi iniziale.

La chiave 'Conflict'

Se l'identificatore di Flowgorithm è in conflitto con una delle parole chiave del linguaggio di destinazione, questa chiave viene utilizzata per generare un nuovo identificatore.

Campi	**Contenuti**
{Name}	Identificatore originale di Flowgorithm

Flags	
integer	L'identificatore è di tipo Integer.
real	L'identificatore è di tipo Real.
boolean	L'identificatore è di tipo Boolean.
string	L'identificatore è di tipo String.
None	L'identificatore non è di alcun tipo conosciuto. Ciò accade quando, ad esempio, si è in presenza di un identificatore non dichiarato. Suggerito come impostazione di sintassi iniziale.

Tipi

La sezione '**Types**' definisce i nomi dei tipi di dato del linguaggio di destinazione.

Esempio Java
[Types] Integer = int Real = double Boolean = boolean String = String

Questa sezione è una tabella semplice. Non ci sono né campi né flags.

Campi
Nessuno

Flags
Nessuno

Gli elementi testuali

I valori stringa testuali

Diversi linguaggi di programmazione possiedono formati ben distinti per le stringhe testuali. Molto spesso, si tratta di una serie di caratteri delimitati da virgolette. Tuttavia, questo non è sempre vero. Alcuni linguaggi usano apice singolo, come Smalltalk e. altri, un formato ancora più particolare.

Ogni volta che il modello deve creare una stringa testuale, utilizza questa sezione. Esistono una serie di tasti di caratteri di sostituzione che sono utili per ignorare i delimitatori.

Esempio Java
[String Literal] Text = "{Characters}" Replace Char 1 = " Replace By 1 = \" Replace Char 2 = \ Replace By 2 = \\

La chiave 'Text'

La chiave **'Text'** viene utilizzata per generare la sintassi della stringa testuale. Il campo **{characters}** contiene una sequenza di caratteri dopo che la stringa originale è stata modificata dalle varie chiavi di sostituzione.

Campi	**Contenuti**
{characters}	I caratteri della stringa letterale

Flags
Nessuno

La chiave 'Replace'

Le stringhe testuali possono contenere fino a 3 coppie di caratteri sostitutivi. Nell'esempio sopra, tutte le occorrenze di **"** saranno sostituite da **\"** e tutte le occorrenze di **** saranno sostituite da ****.

Campi
Nessuno

Flags
Nessuno

I valori interi testuali

I valori interi testuali sono molto presenti nei linguaggi di programmazione. Questa sezione consente di definire in modo esplicito i valori interi testuali.

Esempio Java
[Integer Literal] Text = {integral}

La chiave 'Text'
La chiave **'Text'** viene utilizzata per generare la sintassi di un numero intero.

Campi	**Contenuti**
{integral}	Le cifre che rappresentano la parte intera (integrale) del numero

Flags
Nessuno

I valori reali testuali

I reali testuali sono abbastanza presenti tra la maggior parte dei linguaggi di programmazione, ma tendono a variare da una nazione all'altra. Negli Stati Uniti, il punto '.' viene utilizzato per separare l'intera parte dalla parte frazionaria del numero. In Europa, invece, viene utilizzata la virgola ','.

Questa sezione consente di definire il formato dei numeri reali. Se viene prodotto uno pseudocodice, non esitare a utilizzare il formato della nazione di destinazione.

Esempio Java
[Real Literal] Text = {integral}.{fractional}

La chiave 'Text'
La chiave **'Text'** viene utilizzata per generare la sintassi di un numero reale.

Campi	**Contenuti**
{integral}	Le cifre che rappresentano la parte intera (integrale) del numero
{fractional}	Le cifre che rappresentano la parte frazionaria del numero

Flags
Nessuno

I valori booleani testuali

Questa sezione è una semplice tabella

Esempio Java
[Boolean Literal] True = true False = false

Le chiavi 'True' e 'False'
Queste chiavi definiscono le costanti per 'True' e 'False'.

Campi
Nessuno

Flags
Nessuno

Espressioni

Espressioni e funzioni intrinseche

Le espressioni nei diversi linguaggi di programmazione differiscono notevolmente nella struttura delle precedenze. Di conseguenza, i modelli utilizzano un semplice sistema per convertire qualsiasi espressione Flowgorithm nel formato della lingua di destinazione. Per fare ciò, i modelli usano lo stesso sistema sia per le chiamate di funzioni intrinseche (come Cos, ToInteger, ecc ...) che per gli operatori. Ogni funzione e/o operatore ha gli stessi campi, ovvero {1} e {2}. Questi rappresentano i valori passati ad una funzione intrinseca o la parte a sinistra e la parte a destra di un operatore.

Campi	**Contenuti**
{1}	L'espressione utilizzata nella parte sinistra di un operatore o il primo argomento in una funzione intrinseca.
{2}	L'espressione utilizzata nella parte destra di un operatore o il secondo argomento in una funzione intrinseca.

Livelli di precedenza

Ogni funzione o operatore contiene anche chiavi condivise per la precedenza dell'operatore (nel linguaggio di programmazione di destinazione) e quale precedenza è necessaria per {1} e {2}. Maggiore è il valore numerico assegnato alla funzione o operatore, maggiore è la sua precedenza. Ad esempio, nella maggior parte dei linguaggi di programmazione, la moltiplicazione e la divisione vengono calcolate prima dell'addizione e della sottrazione. In questo caso, gli operatori di moltiplicazione e divisione avranno un valore di precedenza più elevato.

Esempio Java
[Cos] Precedence = 100 Needed 1 = 0 Text = Math.cos({1})

La sezione 'Subexpression'

Ogni volta che il sistema deve incrementare la precedenza di {1} o {2}, si utilizzerà la sezione [**Subexpression**]. Questa sezione differisce dalle altre funzioni e operatori poichè non contiene flag e ha solo un campo {Expression}.

L'esempio di seguito è abbastanza universale per le espressioni in tutti i linguaggi di programmazione.

Esempio Java
[Subexpression] Precedence = 100 Text = ({Expression})

Sezioni e Flags

Sezioni

Le seguenti sezioni usano il formato 'espressione'. Esse contengono le stesse chiavi, flags e la modalità di base.

Sezioni Operatore			
[Add]	[Greater Than]	[Multiply]	[Power]
[And]	[Greater Equal Than]	[Negate]	[Subtract]
[Concatenate]	[Less Than]	[Not]	
[Divide]	[Less Equal Than]	[Not Equals]	
[Equals]	[Modulus]	[Or]	

Sezioni Funzione Intrinseca				
[Abs]	[Cos]	[Pi]	[Sqrt]	[ToInteger]
[ArcCos]	[Int]	[Random]	[Tan]	[ToReal]
[ArcSin]	[Len]	[Sin]	[ToChar]	[ToString]
[ArcTan]	[Log]	[Sgn]	[ToCode]	
[Char]	[Log10]	[Size]	[ToFixed]	

Chiavi

Campi	**Contenuti**
Precedence	Un valore numerico che indica il livello di precedenza di questa funzione / operatore.
Type	Se il tipo di dati, creato da questa funzione / operatore, differisce da Flowgorithm, questo campo può specificare il nuovo tipo. Ad esempio, in Java, 1/2 restituisce un intero. Flowgorithm restituisce un real. Valori consentiti sono: Integer, Real, String e Boolean.
Needed 1	Il livello di precedenza necessario per {1}. In genere questo è uguale o maggiore del valore specificato nella chiave **Precedence**.
Needed 2	Il livello di precedenza necessario per {2}. In genere questo è uguale o maggiore del valore specificato nella chiave **Precedence**.
Text	La sintassi della funzione / operatore.

L'esempio seguente definisce un operatore logico AND di base in Java. I valori di precedenza necessari per {1} e {2} sono tipici degli operatori da sinistra verso destra che consentono cambiamenti (ad esempio: X && y && z).

Esempio Java – Operatore AND
[And] Precedence = 2 Needed 1 = 2 Needed 2 = 3 Text = {1} && {2}

Flags

Diversi linguaggi di programmazione utilizzano spesso operatori diversi o chiamate di librerie, in base al tipo di dato utilizzato. A volte un operatore è incorporato direttamente in un linguaggio di programmazione mentre, in altri casi, richiede una chiamata alla libreria. Un buon esempio di questo è l'operatore esponente. In Flowgorithm e nella famiglia di linguaggio BASIC, il simbolo ^ viene utilizzato per indicare un esponente. I linguaggi della famiglia C (come Java) tendono a utilizzare una chiamata di funzione di una specifica libreria, cioè verrà utilizzata la funzione **Pow()**.

Inoltre, i linguaggi di programmazione hanno regole diverse che definiscono quale tipo di dati viene restituito da un calcolo. Queste varie "conversioni aritmetiche generiche" possono variare notevolmente tra i vari linguaggi. Ad esempio, nella famiglia di linguaggi C l'espressione "1/2" restituirà zero. Le regole stabiliscono che se entrambi gli operandi sono interi, viene utilizzato l'intero matematico. In Flowgorithm e nella famiglia di linguaggi tipo BASIC, la virgola mobile viene sempre utilizzata per la divisione (0,5).

Quindi, per gestire tutti questi diversi scenari, i modelli contengono un numero elevato di flag in modo da poter selezionare la sintassi corretta.

L'esempio seguente definisce un operatore base di addizione in Java. La precedenza necessaria per {1} e {2} è tipica degli operatori da sinistra verso destra che consentono il concatenamento (ad esempio 1 + 2 + 3 + 4). La chiave **Type** è definita per seguire le "conversioni aritmetiche generiche" di Java.

Esempio Java – Operatore ADD
[Add] Type = integer | integer-integer = real | ~integer-integer Precedence = 5 Needed 1 = 5 Needed 2 = 6 Text = {1} + {2}

I seguenti flag permettono di verificare una certa combinazione tra {1} e {2}. Spesso questi sono usati con il prefisso di negazione '~' per gestire casi speciali.

Combinazione Flag	**Da utilizzare quando ….**
string-string	{1} è una stringa, {2} è una stringa
string-integer	{1} è una stringa, {2} è un valore intero
string-real	{1} è una stringa, {2} è un valore reale
string-boolean	{1} è una stringa, {2} è un valore booleano
integer-string	{1} è un valore intero, {2} è una stringa
integer-integer	{1} è un valore intero, {2} è un valore intero
integer-real	{1} è un valore intero, {2} è un valore reale
integer-boolean	{1} è un valore intero, {2} è un valore booleano
real-string	{1} è un valore reale, {2} è una stringa
real-integer	{1} è un valore reale, {2} è un valore intero
real-real	{1} è un valore reale, {2} è un valore reale

real-boolean	{1} è un valore reale, {2} è un valore booleano
boolean-string	{1} è un valore booleano, {2} è una stringa
boolean-integer	{1} è un valore booleano, {2} è una stringa
boolean-real	{1} è un valore booleano, {2} è una stringa
boolean-boolean	{1} è un valore booleano, {2} è una stringa

L'esempio seguente mostra in che modo Java considera le stringhe e i valori numerici in modo diverso. Nota: la precedenza "**Needed**" varia notevolmente in base al confronto di una stringa con un'altra stringa. Nell'ultima riga, quando viene usato **.equals**, {1} deve avere una precedenza di 100 (max in questo modello) e {2} richiede solo 1 (dato che è racchiuso tra parentesi).

Esempio Java

```
[Equals]
Precedence = 3                  | ~string-string
           = 100                | string-string
Needed 1   = 4                  | ~string-string
           = 100                | string-string
Needed 2   = 4                  | ~string-string
           = 1                  | string-string
Text       = {1} == {2}         | ~string-string
           = {1}.equals({2})    | string-string
```

Il tipo di {1} e {2} può anche essere impostato separatamente.

Flag	**Da utilizzare quando ….**
string-1	{1} è una stringa
integer-1	{1} è un intero
real-1	{1} è un reale
boolean-1	{1} è un booleano
string-2	{2} è una stringa
integer-2	{2} è un intero
real-2	{2} è un reale
boolean-2	{2} è un booleano

Accesso alla variabile

Questa sezione viene utilizzata per ottenere la sintassi corretta quando si accede a una variabile in un'espressione.

Esempio Java

```
[Variable Access]
Precedence = 100
Text       = {name}                 | ~subscript
           = {name}[{subscript}]    | subscript
```

La chiave 'Precedence'

Questa chiave definisce la precedenza dell'accesso alla variabile. Normalmente, questa dovrebbe essere impostata sul valore massimo utilizzato nelle espressioni del modello.

La chiave 'Text'

La chiave **Text** viene utilizzata per generare la sintassi per un accesso alla variabile.

Campi	**Contenuti**
{name}	Il nome della variabile
{subscript}	Il simbolo pedice (indice) se si accede a un elemento dell'array.

Il primo e l'ultimo flag possono essere utilizzati se la sintassi differisce nella prima o ultima istruzione nel blocco.

Combinazione Flag	**Da utilizzare quando ….**
subscript	Un simbolo pedice (indice) viene utilizzato con la variabile

Sezione 'Argument'

La sezione degli argomenti viene utilizzata per generare un elenco di espressioni da utilizzare per le chiamate di funzione e le dichiarazioni di chiamata.

Esempio Java
[Argument] Separator = ,{space} Text = {expression}

La chiave 'Separator'

La chiave **'Separator'** definisce il testo che verrà inserito tra l'elenco dei nomi degli argomenti.

Campi
Nessuno

Flags
Nessuno

La chiave 'Text'

La chiave '**Text**' è usata per generare la sintassi della lista degli argomenti.

Campi	**Contenuti**
{Expression}	L'espressione per l'argomento

{Parameter Name}	Il nome del parametro della funzione che corrisponde all'argomento. Questo è necessario per "argomenti nominati" usati in linguaggio come Swift e Smalltalk.

Il primo e l'ultimo flag possono essere utilizzati se la sintassi differisce nella prima o ultima istruzione nel blocco.

Flag	**Da utilizzare quando ….**
First	L'istruzione è il primo elemento nell'elenco.
last	L'istruzione è l'ultimo elemento nell'elenco.

Funzioni

Funzione Principale

La sezione **Main** (funzione) segna l'inizio dell'esecuzione del programma. La sintassi esatta di questa funzione può essere molto utile tra i linguaggi di programmazione. A volte, il blocco delle istruzioni viene lasciato fuori dalle definizioni delle funzioni e, altre volte, i parametri della funzione principale differiscono dal resto del programma.

Esempio Java
[Main] Text = public static void main(String[] args) {{ = -->BLOCK | | 1 = }

La chiave 'Text'
La chiave 'Text' viene utilizzata per generare la sintassi dell'output.

Campi
Nessuno

Il primo e l'ultimo flag possono essere utilizzati se la sintassi differisce per la prima o l'ultima istruzione nel blocco.

Flags
Nessuno (tranne che per i Flags Globali)

Per inserire il blocco della struttura While, si utilizza una singola riga contenente i seguenti codici speciali. Se si vuole vuole cambiare l'indentazione del blocco, assicurasi di specificare l'indentazione dopo il secondo simbolo pipe '|'.

Valore speciale	Cosa fa
-->BLOCK	Inserisce il codice generato dal blocco della funzione.

Parametri

La sezione **Parameter** è usata per generare una lista di parametri da usare nelle sezioni [**Function**] e [**Function Header**].

Esempio Java
[Parameter] Separator = ,{space} Text = {type} {name} | ~array = {type}[] {name} | array

La chiave 'Separator'

La chiave **Separator** definisce il testo che verrà inserito tra l'elenco dei nomi degli argomenti.

Campi
Nessuno

Flags
Nessuno

La chiave 'Text'

La chiave di testo è usata per generare la sintassi della lista degli argomenti.

Campi	Contenuti
{Type}	Il tipo di dati del parametro.
{Name}	Il nome del parametro.

Il primo e l'ultimo flag possono essere utilizzati se la sintassi differisce per la prima o l'ultima istruzione nel blocco.

Flag	Da utilizzare quando ….
First	L'istruzione è il primo elemento nell'elenco.
last	L'istruzione è l'ultimo elemento nell'elenco.

Funzioni

I modelli contengono diverse sezioni per creare la sintassi per le funzioni. Questi includono tutte le funzioni ad eccezione della funzione **Main**. A volte la funzione principale richiede un formato molto specifico o dichiarazioni speciali (per impostare il programma). **Main** è definito in un'altra sezione.

Esempio Java

```
[Function]
Text = public static {type} {name}({parameters}) {{ | return
     = public static void {name}({parameters}) {{   | ~return
     = -->BLOCK                                     |          | 1
     =                                              | return   | 1
     = return {return};                             | return   | 1
     = }
```

La chiave 'Text'

La chiave '**Text'** viene utilizzata per generare la sintassi della dichiarazione di funzione.

Campi	**Contenuti**
{Type}	Il tipo di dato restituito dalla funzione.
{Name}	Il nome della funzione.
{Parameters}	Il testo creato dalla sezione [Parameter]
{Return}	Il nome della variabile restituita

Flag	**Da utilizzare quando ….**
declare	La funzione necessita che vengano dichiarate delle variabili. Questo è utile se il linguaggio di programmazione di destinazione richiede una sezione '**Variable Header**' (ad esempio il linguaggio di programmazione **Pascal**)
parameters	La funzione ha 1 o più parametri
return	La funzione restituisce un valore

Per inserire il blocco della funzione (corpo della funzione), si usa una riga singola contenente i seguenti codici speciali. Se si vuole cambiare l'indentazione del blocco, assicurarsi di specificare l'indentazione dopo il secondo carattere pipe '|'.

Valore speciale	**Cosa fa**
-->BLOCK	Inserisce il codice generato dal blocco della funzione.
-->VARIABLES	Inserisce il codice generato dalla zona **Variable Headers.**

Intestazione di Funzione

Alcuni linguaggi di programmazione richiedono che le funzioni vengano dichiarate all'inizio di un programma. Questo è il caso dei linguaggi di programmazione Pascal e C++.

La sezione **[Funcion Header]** utilizza il testo generato dalla sezione **[Parameter]**. È usata anche nella sezione **[Function]**.

Esempio C++
[Function Header] Text = {type} {name}({parameters}); | return = void {name}({parameters}); | ~return

La chiave 'Text'
La chiave **'Text'** viene utilizzata per generare la sintassi della dichiarazione di funzione.

Campi	**Contenuti**
{Type}	Il tipo di dato restituito dalla funzione.
{Name}	Il nome della funzione.
{Parameters}	Il testo creato dalla sezione [Parameter]
{Return}	Il nome della variabile restituita

Flag	**Da utilizzare quando ….**
declare	La funzione necessita che vengano dichiarate delle variabili. Questo è utile se il linguaggio di programmazione di destinazione richiede una sezione '**Variable Header**' (ad esempio il linguaggio di programmazione **Pascal**)
parameters	La funzione ha 1 o più parametri
return	La funzione restituisce un valore

Intestazione di Variabile

Alcuni linguaggi di programmazione richiedono che le variabili vengano dichiarate all'inizio della funzione e/o del programma. In questi linguaggi esiste una chiara distinzione tra la dichiarazione delle variabili e codice che fa uso di queste ultime. Questo è il caso, ad esempio, del linguaggio di programmazione Pascal.

La sezione **Variable Header** usa lo stesso metodo utilizzato per generare un elenco di variabili. Vengono inserite nelle funzioni usando il valore speciale **---> VARIABLES**.

Esempio Pascal
[Variable Header Name] Separator = ,{space} Text = {name}

[Variable Header] Text = {variables} : array of {type}; | array = {variables} : {type}; ~array

La sezione 'Variable Header Name'

Questa sezione viene utilizzata per generare un elenco di nomi di variabili che verranno utilizzati, in seguito, nella sezione **[Variable Header]**.

La chiave 'Separator'

La chiave definisce il testo che verrà inserito tra l'elenco dei nomi delle variabili.

Campi
Nessuno

Flags
Nessuno

La chiave 'Text'

La chiave **'Text'**viene utilizzata per generare la sintassi di ciascun elemento nell'elenco. La variabile {size} contiene un valore valido se l'istruzione Declare è un Array. Utilizzare il flag 'array' per una corretta sintassi.

Campi	**Contenuti**
{Type}	Il tipo di dato della variabile.
{Name}	Il nome della variabile.
{Size}	La grandezza dell'array.

Il primo e l'ultimo Flag possono essere utilizzati se la sintassi differisce per il primo o l'ultimo elemento nell'elenco.

Flag	**Da utilizzare quando ….**
array	Nella dichiarazione si vuole utilizzare un array
First	L'istruzione è il primo elemento nell'elenco.
last	L'istruzione è l'ultimo elemento nell'elenco.

La sezione 'Declare'

La sezione **'Declare**' viene utilizzata per creare la sintassi per la dichiarazione.

La chiave 'Text'

La chiave **'Text'** viene utilizzata per generare la sintassi di ciascun elemento nell'elenco. La variabile {size} contiene un valore valido se l'istruzione Declare è un array. Utilizzare il flag 'array' per una corretta sintassi.

Campi	Contenuti
{Type}	Il tipo di dato della variabile.
{Name}	Il nome della variabile.
{Size}	La grandezza dell'array.

Il primo e l'ultimo Flag possono essere utilizzati se la sintassi differisce per il primo o l'ultimo elemento nell'elenco.

Flag	Da utilizzare quando ….
array	Nella dichiarazione si vuole utilizzare un array
First	L'istruzione è il primo elemento nell'elenco.
last	L'istruzione è l'ultimo elemento nell'elenco.

La chiave 'Name Mode'

La chiave **'Name Mode'** può contenere due valori diversi:

1. Separate (predefinita)
2. Merged

Quando è impostata a Merged, tutti i nomi delle variabili saranno uniti in un'unica lista. In questo caso, i campi {size} e {type} non hanno senso.

Campi
Nessuno

Flags
Nessuno

Istruzioni

Assegnazione

I modelli utilizzano una singola sezione per definire la sintassi delle dichiarazioni di assegnazione. La sintassi dell'espressione assegnata è definita dalle sezioni operatore / funzione.

Esempio Java
[Assign] Text = {Variable} = {Expression};

La chiave 'Text'

La chiave **'Text'** viene utilizzata per generare la sintassi di ciascun elemento nell'elenco.

Campi	**Contenuti**
{variable}	Nome della variabile. Questa è definita nella sezione [Variable Access].
{expression}	L'espressione. La sintassi è generata dalle varie sezioni di funzione o di espressione.

Il primo e l'ultimo Flag possono essere utilizzati se la sintassi differisce per il primo o l'ultimo elemento nell'elenco.

Flag	**Da utilizzare quando ….**
array	Nella dichiarazione si vuole utilizzare un array
First	L'istruzione è il primo elemento nel blocco.
last	L'istruzione è l'ultimo elemento nel blocco.

Chiamata

I modelli utilizzano una singola sezione per definire la sintassi delle istruzioni di chiamata.

Esempio Java
[Call] Text = {Name}({Arguments});

La chiave 'Text'

La chiave **'Text'** viene utilizzata per generare la sintassi dell'istruzione Output.

Campi	**Contenuti**
{Name}	Nome della funzione.
{Arguments}	Elenco creato utilizzando la sezione **[Argument]**. Vuoto se la chiamata non contiene argomenti. Utilizza il flag **Arguments** per la individuazione.

Il primo e l'ultimo Flag possono essere utilizzati se la sintassi differisce per il primo o l'ultimo elemento nell'elenco.

Flag	Da utilizzare quando ….
array	Nella dichiarazione si vuole utilizzare un array
First	L'istruzione è il primo elemento nel blocco.
last	L'istruzione è l'ultimo elemento nel blocco.

Commenti

I commenti hanno un ruolo essenziale nella documentazione di un programma. La sintassi dei commenti varia notevolmente tra i linguaggi di programmazione con la presenza del commento di riga (in liea) o la presenza del blocco commento.

I blocchi di commento possono causare problemi se il commento contiene l'elemento sintattico che termina il blocco. Di conseguenza, i commenti dei modelli supportano anche la logica di sostituzione dei caratteri della stringa testuale.

Esempio Java
[Comment] Text = // {Text}

La chiave 'Text'

La chiave **'Text'** viene utilizzata per generare la sintassi della stringa testuale. Il campo {text} contiene una sequenza di caratteri dopo che la stringa originale è stata modificata dalle varie chiavi di sostituzione.

Campi	Contenuti
{text}	I caratteri del commento

Flags
Nessuno

La chiave 'Replace'

Le stringhe testuali possono contenere fino a 3 coppie di caratteri sostitutivi. Ad esempio, in Pascal, i commenti di blocco iniziano con un simbolo '{' e terminano con '}'. Nell'esempio seguente, tutte le occorrenze di '}' saranno sostituite da nulla (cioè verranno rimosse).

Esempio Pascal
[Comment] Text = {{ {Text} } Replace Char 1 = } Replace By 1 =

Campi
Nessuno

Flags
Nessuno

Dichiarazione

Esistono due sezioni utilizzate per creare la sintassi per le istruzioni di dichiarazione. Questi definiscono la sintassi della lista dei nomi delle variabili (dichiarati) e la sintassi della dichiarazione stessa.

Esempio Java		
[Declare Name]		
Separator = ,{space}		
Text	= {name}	| ~array
	= {name} = new {Type}[{Size}]	| array
[Declare]		
Text	= {Type} {Variables};	| ~array
	= {Type}[] {Variables};	| array

La sezione 'Declare Name'

Questa sezione viene utilizzata per generare un elenco di nomi di variabili che verranno utilizzati, in seguito, nella sezione **[Declare]**.

La chiave 'Separator'

Campi
Nessuno

Flags
Nessuno

La chiave 'Text'

Campi	Contenuti
{type}	Il tipo del dato della variabile
{name}	Il nome della variabile
{size}	La grandezza dell'array

Il primo e l'ultimo Flag possono essere utilizzati se la sintassi differisce per il primo o l'ultimo elemento nell'elenco.

Flag	Da utilizzare quando ….
array	Nella dichiarazione si vuole utilizzare un array
First	L'istruzione è il primo elemento nel blocco.
last	L'istruzione è l'ultimo elemento nel blocco.

La sezione 'Declare'

La sezione **'Declare'** viene utilizzata per creare la sintassi per un'istruzione di dichiarazione.

La chiave 'Text'

La chiave **'Text'** viene utilizzata per generare la sintassi di ciascun elemento nell'elenco. La variabile **{size}** contiene un valore valido se l'istruzione Declare è un array. Usare il flag 'array' per una corretta sintassi.

Campi	Contenuti
{variables}	I nomi delle variabili generati nella sezione **[Declare Name]**
{type}	Il nome del tipo di dato
{size}	La grandezza dell'array

Il primo e l'ultimo Flag possono essere utilizzati se la sintassi differisce per il primo o l'ultimo elemento nell'elenco.

Flag	Da utilizzare quando ….
array	Nella dichiarazione si vuole utilizzare un array
First	L'istruzione è il primo elemento nel blocco.
last	L'istruzione è l'ultimo elemento nel blocco.

Struttura 'Do'

I modelli usano una singola sezione per definire la sintassi per la struttura **Do**. Si utilizza un codice speciale per indicare dove viene inserito il blocco per la struttura **Do**.

Le seguenti definizioni provengono da una versione semplificata del modello **Java Programming Language**. Notare che ognuno dei sotto blocchi aumenta l'indentazione di 1.

Esempio Java
[Do] Text = do {{ = -->BLOCK | | 1 = } while({condition});

La chiave 'Text'

La chiave **'Text'** viene utilizzata per generare la sintassi dell'istruzione Output.

Campi	Contenuti
{condition}	L'espressione condizionale. La sintassi è generata dalle varie sezioni funzione / espressione.

Il primo e l'ultimo Flag possono essere utilizzati se la sintassi differisce per il primo o l'ultimo elemento nell'elenco.

Flag	Da utilizzare quando ….
block	Il valore speciale contiene uno o più elementi
First	L'istruzione è il primo elemento nel blocco.
last	L'istruzione è l'ultimo elemento nel blocco.

Per inserire il blocco della struttura **Do**, si utilizza una riga singola contenente i seguenti valori speciali. Se si vuole cambiare l'indentazione del blocco, assicurarsi di specificare l'indentazione dopo il secondo simbolo pipe '|'.

Valore speciale	Cosa fa
-->BLOCK	Inserisce il codice generato dal blocco per la struttura **Do**.

Struttura 'For'

I modelli usano una singola sezione per definire la sintassi per la struttura **For**. Si utilizza un codice speciale per indicare dove viene inserito il blocco per la struttura **For**.

Le seguenti definizioni provengono da una versione semplificata del modello **Java Programming Language**. Notare che ognuno dei sotto blocchi aumenta l'indentazione di 1.

Esempio Java

```
[For]
Text = for ({Variable} = {Start}; {Variable} <= {End}; {Variable}++) {{          | inc, ~step
     = for ({Variable} = {Start}; {Variable} <= {End}; {Variable} += {step}) {{ | inc, step
     = for ({Variable} = {Start}; {Variable} >= {End}; {Variable}--) {{          | ~inc, ~step
     = for ({Variable} = {Start}; {Variable} >= {End}; {Variable} -= {step}) {{ | ~inc, step
     = -->BLOCK                                                                 |              | 1
     = end for
```

La chiave 'Text'
La chiave **'Text'** viene utilizzata per generare la sintassi dell'istruzione Output.

Campi	Contenuti
{Variable}	Il nome della variabile che interviene nel ciclo
{Start}	Valore iniziale del ciclo
{End}	Valore finale del ciclo

{Step}	Valore di incrementale del ciclo. Normalmente viene utilizzato l'incremento pari a 1. **Nota: non si può utilizzare un valore negativo. Usare il flag "inc" per determinare la direzione del ciclo.**

Il primo e l'ultimo Flag possono essere utilizzati se la sintassi differisce per il primo o l'ultimo elemento nell'elenco.

Flag	**Da utilizzare quando ….**
Inc	Si incrementa il ciclo (verso una direzione positiva)
step	Il valore di incremento non è uguale a 1
block	Il valore speciale contiene uno o più elementi
First	L'istruzione è il primo elemento nel blocco.
last	L'istruzione è l'ultimo elemento nel blocco.

Per inserire il blocco della struttura **For**, si utilizza una riga singola contenente i seguenti valori speciali. Se si vuole cambiare l'indentazione del blocco, assicurarsi di specificare l'indentazione dopo il secondo simbolo pipe '|'.

Valore speciale	**Cosa fa**
-->BLOCK	Inserisce il codice generato dal blocco per la struttura **For**.

Struttura 'While'

I modelli usano una singola sezione per definire la sintassi per la struttura **While**. Si utilizza un codice speciale per indicare dove viene inserito il blocco per la struttura **While**.

Le seguenti definizioni provengono da una versione semplificata del modello **Java Programming Language**. Notare che ognuno dei sotto blocchi aumenta l'indentazione di 1.

Esempio Java
[While] Text = while({condition}) {{ = -->BLOCK | | 1 = }

La chiave 'Text'
La chiave **'Text'** viene utilizzata per generare la sintassi dell'istruzione Output.

Campi	**Contenuti**
{condition}	L'espressione condizionale. La sintassi è generata dalle varie sezioni funzione / espressione.

Il primo e l'ultimo Flag possono essere utilizzati se la sintassi differisce per il primo o l'ultimo elemento nell'elenco.

Flag	Da utilizzare quando ….
Block	Il valore speciale contiene uno o più elementi
First	L'istruzione è il primo elemento nel blocco.
Last	L'istruzione è l'ultimo elemento nel blocco.

Per inserire il blocco della struttura **While**, si utilizza una riga singola contenente i seguenti valori speciali. Se si vuole cambiare l'indentazione del blocco, assicurarsi di specificare l'indentazione dopo il secondo simbolo pipe '|'.

Valore speciale	Cosa fa
-->BLOCK	Inserisce il codice generato dal blocco per la struttura While.

Struttura 'If'

I modelli usano una singola sezione per definire la sintassi della struttura **If**. Usa due valori speciali per inserire sia il blocco per il vero sia il blocco per il falso (**else**).

Le seguenti definizioni provengono da una versione semplificata del modello **Java Programming Language**. Notare che ognuno dei sotto blocchi aumenta l'indentazione di 1.

Esempio Java
[Assign] Text = if ({condition}) {{ = -->TRUEBLOCK | | 1 = } else {{ | else = -->FALSEBLOCK | else | 1 = }

La chiave 'Text'
La chiave **'Text'** viene utilizzata per generare la sintassi dell'istruzione Output.

Campi	Contenuti
{condition}	L'espressione condizionale. La sintassi è generata dalle varie sezioni funzione / espressione.

Il primo e l'ultimo Flag possono essere utilizzati se la sintassi differisce per il primo o l'ultimo elemento nell'elenco.

Flag	Da utilizzare quando ….
block	Il valore speciale contiene uno o più elementi

else	La struttura **If** ha un blocco **False** (**Else**). In altre parole, il ramo "**else**" del diagramma di flusso contiene forme.
First	L'istruzione è il primo elemento nel blocco.
Last	L'istruzione è l'ultimo elemento nel blocco.

Per inserire il blocco della struttura **If**, si utilizza una riga singola contenente i seguenti valori speciali. Se si vuole cambiare l'indentazione del blocco, assicurarsi di specificare l'indentazione dopo il secondo simbolo pipe '|'.

Valore speciale	**Cosa fa**
-->TRUEBLOCK	Inserisce il codice generato dal blocco **True** della struttura **If**.
-->FALSEBLOCK	Inserisce il codice generato dal blocco **False** (**else**) della struttura **If**.

Input

I modelli usano una singola sezione per definire la sintassi dell'istruzione **Input**.

Esempio Java
[Input] Text = {Variable} = input.nextInt(); | integer = {Variable} = input.nextDouble(); | real = {Variable} = input.nextBoolean(); | boolean = {Variable} = input.nextLine(); | string = {Variable} = input.nextLine(); | none

La chiave 'Text'

La chiave **'Text'** viene utilizzata per generare la sintassi dell'istruzione **Input**. I vari flag possono essere utilizzati per selezionare diverse sintassi in base al tipo di dati letto.

Campi	**Contenuti**
{variable}	I nomi delle variabili generati nella sezione **[Variable Access]**
{type}	Il nome del tipo di dato

Il primo e l'ultimo Flag possono essere utilizzati se la sintassi differisce per il primo o l'ultimo elemento nell'elenco.

Flag	**Da utilizzare quando ….**
integer	La variabile è di tipo Integer.
real	La variabile è di tipo Real.
boolean	La variabile è di tipo Boolean.
string	La variabile è di tipo String.
none	La variabile non è di un tipo conosciuto. Questo è il valore predefinito.

array	La variabile è un array. Alcuni linguaggi di programmazione richiedono sintassi differenti per l'assegnazione di un elemento ad un array.
first	L'istruzione è il primo elemento nel blocco.
last	L'istruzione è l'ultimo elemento nel blocco.

Output

I modelli usano una singola sezione per definire la sintassi dell'istruzione **Output**.

Esempio Java
[Assign] Text = System.out.println({Expression}); | newline = System.out.print({Expression}); | ~newline

La chiave 'Text'

La chiave **'Text'** viene utilizzata per generare la sintassi dell'istruzione **Output**. I vari flag possono essere utilizzati per selezionare diverse sintassi in base al tipo di dati letto.

Campi	**Contenuti**
{expression}	L'espressione. La sintassi è generata dalle varie sezioni di funzione o di espressione.

Il primo e l'ultimo Flag possono essere utilizzati se la sintassi differisce per il primo o l'ultimo elemento nell'elenco.

Flag	**Da utilizzare quando ….**
newline	L'istruzione contempla un ritorno a capo (newline). Flag predefinito nell'istruzione di **Output**
first	L'istruzione è il primo elemento nel blocco.
last	L'istruzione è l'ultimo elemento nel blocco.

Esempi di Paradigma di Programmazione

Linguaggio di programmazione Lua

```
[Language]
Name           = Lua
Extension      = lua
Keywords       = and, break , do, else, elseif, end, false, for, function, if,
               = in, local, nil, not, or,  repeat, return, then, true, until,
while

Conflicts      = index
Case Sensitive = true
Options        =

; ======================================================
; Literals
; ======================================================

[Types]
Integer      =  number
Real         =  number
Boolean      =  boolean
String       =  string

[Function ID]
Convention = camel
Normal     = {Name}
Conflict   = func_{Name}

[Variable ID]
Convention = camel
Normal     = {Name}
Conflict   = var_{Name}

[String Literal]
Text           = "{Characters}"
Replace Char 1 = "
Replace By 1   = \"
Replace Char 2 = \
Replace By 2   = \\

[Boolean Literal]
true     =  true
false    =  false

[Integer Literal]
Text     = {Integral}

[Real Literal]
Text     = {Integral}.{Fractional}

[Variable Access]
Precedence = 100
Text       = {Name}                | ~subscript
           = {Name}[{Subscript}]   | subscript

; ======================================================
; Expressions
; ======================================================
```

```
; Lua precedence:
; 1. or
; 2. and
; 3. Relational ==, >, < ...
; 4. Concatenation: ..
; 5. Addition
; 6. Multiply
; 7. Unary: Not, -
; 8. ^
; 100. Atom, paranthesis

[Or]
Precedence   = 1
Needed 1     = 1
Needed 2     = 2
Text         = {1} or {2}

[And]
Precedence   = 2
Needed 1     = 2
Needed 2     = 3
Text         = {1} and {2}

[Equals]
Precedence   = 3
Needed 1     = 4
Needed 2     = 4
Text         = {1} == {2}

[Not Equals]
Precedence   = 3
Needed 1     = 4
Needed 2     = 4
Text         = {1} ~= {2}

[Less Than]
Precedence   = 3
Needed 1     = 4
Needed 2     = 4
Text         = {1} < {2}

[Less Equal Than]
Precedence   = 3
Needed 1     = 4
Needed 2     = 4
Text         = {1} <= {2}

[Greater Than]
Precedence   = 3
Needed 1     = 4
Needed 2     = 4
Text         = {1} > {2}

[Greater Equal Than]
Precedence   = 3
Needed 1     = 4
Needed 2     = 4
Text         = {1} >= {2}

[Concatenate]
Precedence   = 4
Needed 1     = 4
```

```
Needed 2     = 5
Text         = {1} .. {2}

[Add]
Precedence   = 5
Needed 1     = 5
Needed 2     = 6
Text         = {1} + {2}

[Subtract]
Precedence   = 5
Needed 1     = 5
Needed 2     = 6
Text         = {1} - {2}

[Multiply]
Precedence   = 6
Needed 1     = 6
Needed 2     = 7
Text         = {1} * {2}

[Divide]
Precedence   = 6
Needed 1     = 6
Needed 2     = 7
Text         = {1} / {2}

[Modulus]
Precedence   = 1
Needed 1     = 1
Needed 2     = 1
Text         = math.fmod({1}, {2})

[Power]
Precedence   = 8
Needed 1     = 9
Needed 2     = 8
Text         = {1} ** {2}

[Not]
Precedence   = 7
Needed 1     = 7
Text         = not {1}

[Negate]
Precedence   = 7
Needed 1     = 7
Text         = -{1}

[Subexpression]
Precedence   = 100
Text         = ({Expression})

; ======================================================
; Intrinsic Functions
; ======================================================

[Abs]
Precedence = 100
Needed 1   = 100
Text       =  math.abs{1}
```

```
[ArcCos]
Precedence = 100
Needed 1   = 0
Text       = math.acos({1})

[ArcSin]
Precedence = 100
Needed 1   = 0
Text       = math.asin({1})

[ArcTan]
Precedence = 100
Needed 1   = 0
Text       = math.atan({1})

[Char]
Precedence = 100
Needed 1   = 100
Needed 2   = 0
Text       = string.sub({1}, index({1}), index({1})

[Cos]
Precedence = 100
Needed 1   = 0
Text       = math.cos({1})

[Int]
Precedence = 100
Needed 1   = 100
Text       =  math.floor({1})

[Len]
Precedence = 100
Needed 1   = 100
Text       = string.len({1})

[Log]
Precedence = 100
Needed 1   = 0
Text       = math.log({1})

[Log10]
Precedence = 100
Needed 1   = 0
Text       = math.log10({1})

[Pi]
Precedence = 100
Text       = math.pi

[Random]
Precedence = 100
Needed 1   = 0
Text       = (math.random({1}) - 1)

[Sin]
Precedence = 100
Needed 1   = 0
Text       = math.sin({1})

[Sgn]
Precedence = 100
```

```
Needed 1   = 0
Text       = sgn({1})

[Size]
Precedence = 100
Needed 1   = 100
Text       = size({1})

[Sqrt]
Precedence = 100
Needed 1   = 0
Text       = math.sqrt({1})

[Tan]
Precedence = 100
Needed 1   = 0
Text       = math.tan({1})

[ToChar]
Precedence = 100
Needed 1   = 0
Text       = string.char({1})

[ToCode]
Precedence = 100
Needed 1   = 0
Text       = string.byte({1})

[ToInteger]
Precedence = 100
Needed 1   = 0
Text       = math.floor(tonumber({1}))

[ToFixed]
Precedence = 100
Needed 1   = 0
Needed 2   = 0
Text       = toFixed({1}, {2})

[ToReal]
Precedence = 100
Needed 1   = 0
Text       = tonumber({1})

[ToString]
Precedence = 100
Needed 1   = 0
Text       = tostring({1})

; ======================================================
; Function call
; ======================================================

[Function Call]
Precedence = 100
Text       = {name}({arguments})

[Argument]
Separator  = ,{space}
Text       = {expression}

; ======================================================
```

```
; Program
; =====================================================

[Program]
Text      = -- Gets the size of the array (by using the max index)  | size
          = function size(values)                                   | size
          = return table.maxn(values) + 1                           | size | 1
          = end                                                     | size
          =                                                         | size

          = -- Lua lacks a sign function. This code implements it.    | sgn
          = function sgn(n)                                           | sgn
          = if n == 0 then                                            | sgn   |
1
          = return 0                                                  | sgn   |
2
          = elseif n < 0 then                                         | sgn   |
1
          = return -1                                                 | sgn   |
2
          = else                                                      | sgn   |
1
          = return 1                                                  | sgn   |
1
          = end                                                       | sgn   |
1
          = end                                                       | sgn   |
          =                                                           | sgn

          = def toFixed(value, digits)                                | tofixed
          = return string.format("%0." .. digits .. "f", value)       | tofixed
| 1
          = end                                                       | tofixed
          =                                                           | tofixed

          = -->FUNCTIONS
          = -->MAIN

[Main]
Text      = -- Main                          | functions
          = math.randomseed(os.time ())  -- Prepare the random number generator
| random
          =
| random
          = -->BLOCK

; =====================================================
; Function
; =====================================================

[Function]
Text      = function {name}({parameters})
          = -->BLOCK                                          |        | 1
          =                                                   | return | 1
          = return {return}                                   | return | 1
          = end
          =

[Parameter]
Separator = ,{space}
Text      = {name}
```

```
; ========================================================
; Statements
; ========================================================

[Assign]
Text       = {Variable} = {Expression}

[Call]
Text       = {Name} ({Arguments})

[Comment]
Text       = -- {Text}

[Declare Name]
Separator  = ,{space}
Text       = {name}          | ~array
           = {Name} = {{}    | array

[Declare]
Text       = local {variables}

[Do]
Text       = repeat
           = -->BLOCK                                | | 1
           = until not ({condition})

[For]
Text       = for {Variable} = {start}, {end} do              | inc, ~step
           = for {Variable} = {start}, {end}, {step} do      | inc, step
           = for {Variable} = {start}, {end}, -{step} do     | ~inc
           = -->BLOCK                                        |           | 1
           = end

[Input]
Text       = {Variable} = tonumber(io.read())                  | integer
           = {Variable} = tonumber(io.read())                  | real
           = {Variable} = string.lower(io.read()) == "true")   | boolean
           = {Variable} = io.read()                            | string
           = {Variable} = io.read()                            | none

[If]
Text       = if {condition} then
           = -->TRUEBLOCK                  |         | 1
           = else                          | else
           = -->FALSEBLOCK                 | else    | 1
           = end

[Output]
Separator = ,{space}
Text       = io.write({List}, "\n")       | newline
           = io.write({List})             | ~newline

[While]
Text       = while {condition} do
           = -->BLOCK                    | | 1
           = end
```

Linguaggio di programmazione Java (semplificato)

```
[Language]
Name           = Java
Extension      = java
Keywords       = abstract, assert, boolean, break, byte, case, catch, char,
class, const
               = continue, default, double, do, else, enum, extends, false,
final, finally
               = float, for, goto, if, implements, import, instanceof, int,
interface, long
               = native, new, null, package, private, protected, public, return,
short, static
               = strictfp, super, switch, synchronized, this, throw, throws,
transient, true, try
               = void, volatile, while

Conflicts      = input
Case Sensitive = true
Options        =

; ========================================================
; Literals
; ========================================================

[Types]
Integer      = int
Real         = double
Boolean      = boolean
String       = String

[Function ID]
Convention = camel
Normal     = {Name}
Conflict   = func_{Name}

[Variable ID]
Convention = camel
Normal     = {Name}
Conflict   = var_{Name}

[String Literal]
Text           = "{Characters}"
Replace Char 1 = "
Replace By 1   = \"
Replace Char 2 = \
Replace By 2   = \\

[Boolean Literal]
true     =  true
false    =  false

[Integer Literal]
Text     = {Integral}

[Real Literal]
Text     = {Integral}.{Fractional}

[Variable Access]
Precedence = 100
Text       = {Name}              | ~subscript
```

```
           = {Name}[{Subscript}]  | subscript

; ========================================================
; Expressions
; ========================================================
; 1. or
; 2. and
; 3. ==
; 4. Relational >, < ...
; 5. Addition
; 6. Multiply
; 7. Cast ()
; 8. Unary: Not, !
; 100. Atom, paranthesis

[Or]
Precedence   = 1
Needed 1     = 1
Needed 2     = 2
Text         = {1} {pipe}{pipe} {2}

[And]
Precedence   = 2
Needed 1     = 2
Needed 2     = 3
Text         = {1} && {2}

[Equals]
Precedence   = 3                |  ~string-string
             = 100              |  string-string

Needed 1     = 4                |  ~string-string
             = 100              |  string-string

Needed 2     = 4                |  ~string-string
             = 1                |  string-string

Text         = {1} == {2}       |  ~string-string
             = {1}.equals({2})  |  string-string

[Not Equals]
Precedence   = 3                |  ~string-string
             = 100              |  string-string

Needed 1     = 4                |  ~string-string
             = 100              |  string-string

Needed 2     = 4                |  ~string-string
             = 1                |  string-string

Text         = {1} != {2}       |  ~string-string
             = !{1}.equals({2})  |  string-string

[Less Than]
Precedence   = 4

Needed 1     = 5                      |  ~string-string
             = 100                    |  string-string

Needed 2     = 5                      |  ~string-string
             = 1                      |  string-string
```

```
Text         = {1} < {2}                 |  ~string-string
             = {1}.compareTo({2}) < 0    |  string-string

[Less Equal Than]
Precedence   = 4

Needed 1     = 5                         |  ~string-string
             = 100                       |  string-string

Needed 2     = 5                         |  ~string-string
             = 1                         |  string-string

Text         = {1} <= {2}                |  ~string-string
             = {1}.compareTo({2}) <= 0   |  string-string

[Greater Than]
Precedence   = 4

Needed 1     = 5                         |  ~string-string
             = 100                       |  string-string

Needed 2     = 5                         |  ~string-string
             = 1                         |  string-string

Text         = {1} > {2}                 |  ~string-string
             = {1}.compareTo({2}) > 0    |  string-string

[Greater Equal Than]
Precedence   = 4

Needed 1     = 5                         |  ~string-string
             = 100                       |  string-string

Needed 2     = 5                         |  ~string-string
             = 1                         |  string-string

Text         = {1} >= {2}                |  ~string-string
             = {1}.compareTo({2}) >= 0   |  string-string

[Concatenate]
Precedence   = 5

Needed 1     = 5          | string-1
             = 1          | ~string-1
Needed 2     = 6

Text         = {1} + {2}                      | string-1
             = Integer.toString({1}) + {2}    | integer-1
             = Double.toString({1}) + {2}     | real-1
             = Boolean.toString({1}) + {2}    | boolean-1

[Add]
Type         = real         | ~integer-integer
             = integer      | integer-integer

Precedence   = 5
Needed 1     = 5
Needed 2     = 6
Text         = {1} + {2}

[Subtract]
Type         = real         | ~integer-integer
```

```
             = integer      | integer-integer

Precedence   = 5
Needed 1     = 5
Needed 2     = 6
Text         = {1} - {2}

[Multiply]
Type         = real         | ~integer-integer
             = integer      | integer-integer

Precedence   = 6
Needed 1     = 6
Needed 2     = 7
Text         = {1} * {2}

[Divide]
Type         = real
Precedence   = 6
Needed 1     = 6              |  ~integer-integer
             = 100            |  integer-integer

Needed 2     = 7
Text         = {1} / {2}              |  ~integer-integer
             = (double) {1} / {2}     |  integer-integer

[Modulus]
Precedence   = 6
Needed 1     = 6
Needed 2     = 7
Text         = {1} % {2}

[Power]
Type         = real
Precedence   = 100
Needed 1     = 0
Needed 2     = 0
Text         = Math.pow({1}, {2})

[Not]
Type         = boolean
Precedence   = 7
Needed 1     = 7
Text         = !{1}

[Negate]
Type         = real         | ~integer
             = integer      | integer

Precedence   = 7
Needed 1     = 7
Text         = -{1}

[Subexpression]
Precedence   = 100
Text         = ({Expression})

; ========================================================
; Intrinsic Functions
; ========================================================

[Abs]
```

```
Precedence = 100
Needed 1   = 0
Text       = Math.abs({1})

[ArcCos]
Precedence = 100
Needed 1   = 0
Text       = Math.acos({1})

[ArcSin]
Precedence = 100
Needed 1   = 0
Text       = Math.asin({1})

[ArcTan]
Precedence = 100
Needed 1   = 0
Text       = Math.atan({1})

[Char]
Precedence = 100
Needed 1   = 100
Needed 2   = 0
Text       = {1}.charAt({2})

[Cos]
Precedence = 100
Needed 1   = 0
Text       = Math.cos({1})

[Int]
Precedence = 100
Needed 1   = 0
Text       = Math.floor({1})

[Len]
Precedence = 100
Needed 1   = 100
Text       = {1}.length()

[Log]
Precedence = 100
Needed 1   = 0
Text       = Math.log({1})

[Log10]
Precedence = 100
Needed 1   = 0
Text       = Math.log10({1})

[Pi]
Precedence = 100
Text       = Math.PI

[Random]
Precedence = 100
Needed 1   = 0
Text       = random.nextInt({1})

[Sin]
Precedence = 100
Needed 1   = 0
```

```
Text       = Math.sin({1})

[Sgn]
Precedence = 100
Needed 1   = 0
Text       = Math.signum({1})

[Size]
Precedence = 100
Needed 1   = 100
Text       = {1}.length

[Sqrt]
Precedence = 100
Needed 1   = 0
Text       = Math.sqrt({1})

[Tan]
Precedence = 100
Needed 1   = 0
Text       = Math.tan({1})

[ToChar]
Precedence = 100
Needed 1   = 100
Text       = (char) {1}

[ToCode]
Precedence = 100
Needed 1   = 100
Text       = (int) {1}[0]

[ToFixed]
Type       = string
Precedence = 100
Needed 1   = 0
Needed 2   = 0
Text       = toFixed({1},{2})

[ToInteger]
Precedence = 100
Needed 1   = 0
Text       = Integer.parseInt({1})

[ToReal]
Precedence = 100
Needed 1   = 0
Text       = Integer.parseDouble({1})

[ToString]
Precedence = 100
Needed 1   = 100
Text       = {1}.toString()

; ======================================================
; Function call
; ======================================================

[Function Call]
Precedence = 100
Text       = {name}({arguments})
```

```
[Argument]
Separator  = ,{space}
Text       = {expression}

; ========================================================
; Program
; ========================================================

[Program]
Text      = import java.util.*;
          = import java.lang.Math;
          =
          = public class JavaApplication {{
          = private static Random random = new Random();                  |
random            | 1
          = private static Scanner input = new Scanner(System.in);        | input
| 1
          =                                                               |
random, ~input
          =                                                               |
~random, input
          =                                                               |
random, input
          = -->MAIN                                                       |
| 1
          = -->FUNCTIONS                                                  |
| 1
          =                                                               |
tofixed           | 1
          = private static String toFixed(double value, int digits) {{   |
tofixed           | 1
          = return String.format("%." + digits + "f", value);             |
tofixed           | 2
          = }                                                             |
tofixed           | 1
          = }

[Main]
Text      = public static void main(String[] args) {{
          = -->BLOCK                                                      |
| 1
          = }

; ========================================================
; Function
; ========================================================

[Parameter]
Separator = ,{space}
Text      = {type} {name}             | ~array
          = {type}[] {name}           | array

[Function]
Text      =
          = public static {type} {name}({parameters}) {{      | return
          = public static void {name}({parameters}) {{        | ~return
          = -->BLOCK                                          |          | 1
          =                                                   |  return  | 1
          = return {return};                                  |  return  | 1
          = }

; ========================================================
```

```
; Statements
; ========================================================

[Assign]
Text       = {Variable} = {Expression};

[Call]
Text       = {Name}({Arguments});

[Comment]
Text       = // {Text}

[Declare Name]
Separator  = ,{space}
Text       = {name}                              | ~array
           = {name} = new {Type}[{Size}]         | array

[Declare]
Text       = {Type} {Variables};                 | ~array
           = {Type}[] {Variables};               | array

[Do]
Text       = do {{
           = -->BLOCK                            |  | 1
           = } while ({condition});

[For]
Text       = for ({Variable} = {Start}; {Variable} <= {End}; {Variable}++) {{
| inc, ~step
           = for ({Variable} = {Start}; {Variable} <= {End}; {Variable} +=
{step}) {{   | inc, step
           = for ({Variable} = {Start}; {Variable} >= {End}; {Variable}--) {{
| ~inc, ~step
           = for ({Variable} = {Start}; {Variable} >= {End}; {Variable} -=
{step}) {{   | ~inc, step
           = -->BLOCK
|             | 1
           = }

[Input]
Text       = {Variable} = input.nextInt();                                  |
integer
           = {Variable} = input.nextDouble();                               | real
           = {Variable} = input.nextBoolean();                              |
boolean
           = {Variable} = input.nextLine();                                 | string
           = {Variable} = input.nextLine();                                 | none

[If]
Text       = if ({condition}) {{
           = -->TRUEBLOCK                  |        | 1
           = } else {{                     | else
           = -->FALSEBLOCK                 | else   | 1
           = }

[Output]
Text       = System.out.println({Expression});           | newline
           = System.out.print({Expression});             | ~newline

[While]
Text       = while ({condition}) {{
           = -->BLOCK                      |     | 1
```

```
= }
```

Linguaggio di programmazione Pascal

```
[Language]
Name           = Pascal
Extension      = pas

Keywords       = and, array, as, asm, begin, case, class, const, constructor,
destructor
               = dispinterface, div, do, downto, else, end, except, exports,
file, finalization
               = finally, for, function, goto, if, implementation, in,
inherited, initialization, interface
               = in, is, library, nil, not, object, of, or, out, packed
               = procedure, program, property, raise, randomize, record, repeat,
resourcestring, set
               = string, then, to, try, type, unit, until, uses, var, while,
with

Conflicts      =

Case Sensitive = false
Options        =

; ============================================================
; Literals
; ============================================================

[Types]
Integer      = integer
Real         = real
Boolean      = boolean
String       = string

[Function ID]
Convention = proper
Normal     = {Name}
Conflict   = {Name}_function

[Variable ID]
Convention = proper
Normal     = {Name}
Conflict   = {Name}_variable

[String Literal]
Text           = '{Characters}'
Replace Char 1 = '
Replace By 1   = ''

[Boolean Literal]
true     =  true
false    =  false

[Integer Literal]
Text     = {Integral}

[Real Literal]
Text     = {Integral}.{Fractional}

[Variable Access]
Precedence = 100
Text       = {Name}              | ~subscript
```

```
          = {Name}[{Subscript}]  | subscript

; ======================================================
; Expressions
; ======================================================
; Pascal precedence: (ONLY FOUR)
; 1. Comparison:       =, <>, <, >, <=, >=
; 2. Addition & Or:    +, -, or, xor
; 3. Multiply & And:   *, /, mod, and
; 4. Not:              not
; 100. Atom, paranthesis

[Or]
Precedence   = 3
Needed 1     = 3
Needed 2     = 4
Text         = {1} or {2}

[And]
Precedence   = 3
Needed 1     = 3
Needed 2     = 4
Text         = {1} and {2}

[Equals]
Precedence   = 1
Needed 1     = 2
Needed 2     = 2
Text         = {1} = {2}

[Not Equals]
Precedence   = 1
Needed 1     = 2
Needed 2     = 2
Text         = {1} <> {2}

[Less Than]
Precedence   = 1
Needed 1     = 2
Needed 2     = 2
Text         = {1} < {2}

[Less Equal Than]
Precedence   = 1
Needed 1     = 2
Needed 2     = 2
Text         = {1} <= {2}

[Greater Than]
Precedence   = 1
Needed 1     = 2
Needed 2     = 2
Text         = {1} > {2}

[Greater Equal Than]
Precedence   = 1
Needed 1     = 2
Needed 2     = 2
Text         = {1} >= {2}

[Concatenate]
Precedence   = 2
```

```
Needed 1     = 2            | string-1
             = 1            | ~string-1

Needed 2     = 3            | string-2
             = 1            | ~string-2

Text         = {1} + {2}                              | string-1, string-2
             = FloatToStr({1}) + {2}                  | ~string-1, string-2
             = {1} + FloatToStr({2})                  | string-1, ~string-2
             = FloatToStr({1}) + FloatToStr({2})      | ~string-1, ~string-2

[Add]
Precedence   = 2
Needed 1     = 2
Needed 2     = 3
Text         = {1} + {2}

[Subtract]
Precedence   = 2
Needed 1     = 2
Needed 2     = 3
Text         = {1} - {2}

[Multiply]
Precedence   = 3
Needed 1     = 3
Needed 2     = 4
Text         = {1} * {2}

[Divide]
Precedence   = 3
Needed 1     = 3
Needed 2     = 4
Text         = {1} / {2}

[Modulus]
Precedence   = 3
Needed 1     = 3
Needed 2     = 4
Text         = {1} mod {2}

[Power]
Precedence   = 100
Needed 1     = 1
Needed 2     = 1
Text         = Power({1}, {2})

[Not]
Precedence   = 4
Needed 1     = 4
Text         = Not {1}

[Negate]
Precedence   = 4
Needed 1     = 4
Text         = -{1}

[Subexpression]
Precedence   = 100
Text         = ({Expression})

; ======================================================
```

```
; Intrinsic Functions
; ====================================================

[Abs]
Precedence = 100
Needed 1   = 1
Text       = Abs({1})

[ArcCos]
Precedence = 100
Needed 1   = 1
Text       = ArcCos({1})

[ArcSin]
Precedence = 100
Needed 1   = 1
Text       = ArcSin({1})

[ArcTan]
Precedence = 100
Needed 1   = 1
Text       = ArcTan({1})

[Char]
Precedence = 100
Needed 1   = 100
Needed 2   = 1
Text       = {1}[{2}]

[Cos]
Precedence = 100
Needed 1   = 1
Text       = Cos({1})

[Int]
Precedence = 100
Needed 1   = 1
Text       = Int({1})

[Len]
Precedence = 100
Needed 1   = 1
Text       = Length({1})

[Log]
Precedence = 100
Needed 1   = 1
Text       = Log({1})

[Log10]
Precedence = 100
Needed 1   = 1
Text       = Log10({1})

[Pi]
Precedence = 100
Text       = Pi

[Random]
Precedence = 100
Needed 1   = 1
Text       = Random({1})
```

```
[Sin]
Precedence = 100
Needed 1   = 1
Text       = Sin({1})

[Sgn]
Precedence = 100
Needed 1   = 1
Text       = Sgn({1})

[Size]
Precedence = 100
Needed 1   = 1
Text       = Length({1})

[Sqrt]
Precedence = 100
Needed 1   = 1
Text       = Sqrt({1})

[Tan]
Precedence = 100
Needed 1   = 1
Text       = Tan({1})

[ToChar]
Precedence = 100
Needed 1   = 1
Text       = Chr({1})

[ToCode]
Precedence = 100
Needed 1   = 1
Text       = Ord({1})

[ToFixed]
Precedence = 100
Needed 1   = 1
Needed 2   = 1
Text       = ToFixed({1}, {2})

[ToInteger]
Precedence = 100
Needed 1   = 1
Text       = StrToInt({1})

[ToReal]
Precedence = 100
Needed 1   = 1
Text       = StrToFloat({1})

[ToString]
Precedence = 100
Needed 1   = 1
Text       = FloatToStr({1})

; ======================================================
; Function call
; ======================================================

[Function Call]
```

```
Precedence = 100
Text       = {name}({arguments})

[Argument]
Separator  = ,{space}
Text       = {expression}

; ======================================================
; Program
; ======================================================

[Program]
Text      = program MyProgram;
          = uses Math, SysUtils;

          =                                  | functions
          = {{ Headers }                     | functions
          = -->HEADERS                       | functions

          = -->FUNCTIONS
          = -->MAIN

[Main]
Text      =                                  | functions
          = {{ Main }                        | functions
          = var                              | declare
          = -->VARIABLES                     | declare   | 1
          =                                  | declare
          = begin
          = randomize; {{Prepare the random number generator}  | random | 1
          =                                                    | random
          = -->BLOCK                         | | 1
          = end.

; ======================================================
; Function
; ======================================================

[Parameter]
Separator = ,{space}
Text      = {name} : {type}                    | ~array
          = var {name} : array of {type}       | array

[Function]
Block Extra =  1         | return
            =  0         | ~return

Text        =
            = procedure {name} ({parameters});          | ~return
            = var                                       | ~return, declare
            = -->VARIABLES                              | ~return, declare | 1
            =                                           | ~return, declare
            = begin                                     | ~return
            = -->BLOCK                                  | ~return          | 1
            = end;                                      | ~return

            = function {name} ({parameters}) : {type};  | return
            = var                                       | return, declare
            = -->VARIABLES                              | return, declare | 1
            =                                           | return, declare
            = begin                                     | return
            = -->BLOCK                                  | return          | 1
```

```
           =                                                | return          | 1
           = {name} := {return}                             | return          | 1
           = end;                                           | return

[Function Header]
Text       = procedure {name} ({parameters}); forward;         | ~return
           = function {name} ({parameters}) : {type}; forward; | return

; ========================================================
; Variable Header
; ========================================================

[Variable Header Name]
Separator  = ,{space}
Text       = {name}

[Variable Header]
Text       = {variables} : array of {type};    | array
           = {variables} : {type};             | ~array

; ========================================================
; Statements
; ========================================================

[Assign]
Text       = {Variable} := {Expression};    | ~last
           = {Variable} := {Expression}     | last

[Call]
Text       = {Name} ({Arguments});          | ~last
           = {Name} ({Arguments})           | last

[Comment]
Replace Char 1 = }
Replace By 1   =
Text           = {{ {Text} }

[Declare Name]
Separator  = ,{space}
Text       = {name}

[Declare]
Name Mode = Singular
Text       = SetLength({Name}, {Size});       |  array, ~last
           = SetLength({Name}, {Size})        |  array, last

[Do]
Text       = repeat
           = begin
           = -->BLOCK                      | | 1
           = end
           = until not ({condition});      | ~last
           = until not ({condition})       | last

[For]
Text       = For {Variable} := {Start} to {End} do          | ~step, inc
           = For {Variable} := {Start} downto {End} do      | ~step, ~inc
           = begin                                          | ~step
           = -->BLOCK                                       | ~step
| 1
           = end;                                           | ~step, ~last
           = end                                            | ~step, last
```

```
          = {Variable} := {Start};                          | step
          = while {Variable} <= {End} do                    | step, inc
          = while {Variable} >= {End} do                    | step, ~inc
          = begin                                           | step
          = -->BLOCK                                        | step        | 1
          =                                                 | step
          = {Variable} := {Variable} + {Step};              | step, inc
          = {Variable} := {Variable} - {Step};              | step, ~inc
          = end;                                            | step, ~last
          = end                                             | step, last


[Input]
Text      = ReadLn({Variable});           | ~last
          = ReadLn({Variable})            | last
[If]
Text      = if {condition} then
          = begin
          = -->TRUEBLOCK                  |       | 1

          = end                           | else
          = else                          | else
          = begin                         | else
          = -->FALSEBLOCK                 | else  | 1

          = end;                          | ~last
          = end                           | last

[Output]
Text      = WriteLn({Expression});        | newline, ~last
          = Write({Expression});          | ~newline, ~last

          = WriteLn({Expression})         | newline, last
          = Write({Expression})           | ~newline, last

[While]
Text      = while {condition} do
          = begin
          = -->BLOCK                      | | 1

          = end;                          | ~last
          = end                           | last
```

Linguaggio di programmazione Python

```
[Language]
Name           = Python
Extension      = py
Keywords       = and, as, assert, break, class, continue, def, del, elif, else
               = except, exec, finally, for, from, global, if, import, in, is
               = lambda, not, or, pass, print, raise, return, try, while, with
               = yield

               =  True, False, None
Conflicts
Case Sensitive = true
Options        =

; ========================================================
; Literals
; ========================================================

[Types]
Integer      =  int
Real         =  float
Boolean      =  bool
String       =  str

[Function ID]
Convention = camel
Normal     = {Name}
Conflict   = func_{Name}

[Variable ID]
Convention = camel
Normal     = {Name}
Conflict   = var_{Name}

[String Literal]
Text           = "{Characters}"
Replace Char 1 = "
Replace By 1   = \"
Replace Char 2 = \
Replace By 2   = \\

[Boolean Literal]
true     =  True
false    =  False

[Integer Literal]
Text     = {Integral}

[Real Literal]
Text     = {Integral}.{Fractional}

[Variable Access]
Precedence = 100
Text       = {Name}                | ~subscript
           = {Name}[{Subscript}]   | subscript

; ========================================================
; Expressions
; ========================================================
; Python precedence:
```

```
; 1. or
; 2. and
; 3. not
; 4. Relational and equality: =, >, < ...
; 5. Addition
; 6. Multiply
; 7. Unary: -
; 8. **
; 100. Atom, paranthesis

[Or]
Precedence   = 1
Needed 1     = 1
Needed 2     = 2
Text         = {1} or {2}

[And]
Precedence   = 2
Needed 1     = 2
Needed 2     = 3
Text         = {1} and {2}

[Equals]
Precedence   = 3
Needed 1     = 4
Needed 2     = 4
Text         = {1} == {2}

[Not Equals]
Precedence   = 3
Needed 1     = 4
Needed 2     = 4
Text         = {1} != {2}

[Less Than]
Precedence   = 3
Needed 1     = 4
Needed 2     = 4
Text         = {1} < {2}

[Less Equal Than]
Precedence   = 3
Needed 1     = 4
Needed 2     = 4
Text         = {1} <= {2}

[Greater Than]
Precedence   = 3
Needed 1     = 4
Needed 2     = 4
Text         = {1} > {2}

[Greater Equal Than]
Precedence   = 3
Needed 1     = 4
Needed 2     = 4
Text         = {1} >= {2}

[Concatenate]
Precedence   = 5

Needed 1     = 5        | string-1
```

```
             = 1         | ~string-1

Needed 2     = 6         | string-2
             = 1         | ~string-2

Text         = {1} + {2}                 | string-1, string-2
             = {1} + str({2})            | string-1, ~string-2
             = str({1}) + {2}            | ~string-1, string-2
             = str({1}) + str({2})       | ~string-1, ~string-2

[Add]
Type         = real         | ~integer-integer
             = integer      | integer-integer

Precedence   = 5
Needed 1     = 5
Needed 2     = 6
Text         = {1} + {2}

[Subtract]
Type         = real         | ~integer-integer
             = integer      | integer-integer

Precedence   = 5
Needed 1     = 5
Needed 2     = 6
Text         = {1} - {2}

[Multiply]
Type         = real         | ~integer-integer
             = integer      | integer-integer

Precedence   = 6
Needed 1     = 6
Needed 2     = 7
Text         = {1} * {2}

[Divide]
Type         = real
Precedence   = 6
Needed 1     = 6              |  ~integer-integer
             = 1              |  integer-integer

Needed 2     = 7
Text         = {1} / {2}           |  ~integer-integer
             = float({1}) / {2}    |  integer-integer

[Modulus]
Type         = integer
Precedence   = 6
Needed 1     = 6
Needed 2     = 7
Text         = {1} % {2}

[Power]
Type         = real
Precedence   = 8
Needed 1     = 9
Needed 2     = 8
Text         = {1} ** {2}

[Not]
```

```
Type         = boolean
Precedence   = 3
Needed 1     = 3
Text         = not {1}

[Negate]
Type         = real         | ~integer
             = integer      | integer

Precedence   = 7
Needed 1     = 7
Text         = -{1}

[Subexpression]
Precedence   = 100
Text         = ({Expression})

; ======================================================
; Intrinsic Functions
; ======================================================

[Abs]
Precedence = 100
Needed 1   = 0
Text       = fabs({1})

[ArcCos]
Precedence = 100
Needed 1   = 0
Text       = acos({1})

[ArcSin]
Precedence = 100
Needed 1   = 0
Text       = asin({1})

[ArcTan]
Precedence = 100
Needed 1   = 0
Text       = atan({1})

[Char]
Precedence = 100
Needed 1   = 100
Needed 2   = 0
Text       = {1}[{2}]

[Cos]
Precedence = 100
Needed 1   = 0
Text       = cos({1})

[Int]
Precedence = 100
Needed 1   = 0
Text       = int({1})

[Len]
Precedence = 100
Needed 1   = 0
Text       = len({1})
```

```
[Log]
Precedence = 100
Needed 1   = 0
Text       = log({1})

[Log10]
Precedence = 100
Needed 1   = 0
Text       = log10({1})

[Pi]
Precedence = 100
Text       = math.pi

[Random]
Precedence = 100
Needed 1   = 6
Text       = int(random.random() * {1})

[Sin]
Precedence = 100
Needed 1   = 0
Text       = sin({1})

[Sgn]
Precedence = 100
Needed 1   = 0
Text       = sgn({1})

[Size]
Precedence = 100
Needed 1   = 0
Text       = len({1})

[Sqrt]
Precedence = 100
Needed 1   = 0
Text       = sqrt({1})

[Tan]
Precedence = 100
Needed 1   = 0
Text       = tan({1})

[ToChar]
Precedence = 100
Needed 1   = 0
Text       = chr({1})

[ToCode]
Precedence = 100
Needed 1   = 0
Text       = ord({1})

[ToInteger]
Precedence = 100
Needed 1   = 0
Text       = int({1})

[ToFixed]
Type       = string
Precedence = 100
```

```
Needed 1   = 0
Needed 2   = 0
Text       = toFixed({1},{2})

[ToReal]
Precedence = 100
Needed 1   = 0
Text       = float({1})

[ToString]
Precedence = 100
Needed 1   = 0
Text       = str({1})

; ========================================================
; Function call
; ========================================================

[Function Call]
Precedence = 100
Text       = {name}({arguments})

[Argument]
Separator  = ,{space}
Text       = {expression}

; ========================================================
; Program
; ========================================================

[Program]
Text      = import random                                              | random
          =                                                            | random

          = # Python lacks a sign function. This code implements it.   | sgn
          = def sgn(n):                                                | sgn
          = if n == 0: return 0                                        | sgn   |
1
          = elif n < 0: return -1                                      | sgn   |
1
          = else: return 1                                             | sgn   |
1
          =                                                            | sgn

          = def toFixed(value, digits):                                | tofixed
          = return "%.*f" % (digits, value)                            | tofixed
| 1
          =                                                            | tofixed

          = -->FUNCTIONS
          = -->MAIN

[Main]
Text      = # Main                                                     |
functions
          = random.seed()   #Prepare random number generator           | random
          = -->BLOCK

; ========================================================
; Function
; ========================================================
```

```
[Function]
Text      = def {name}({parameters}):
          = -->BLOCK                                              |        | 1
          = pass                                                  | ~block | 1
          =                                                       | return | 1
          = return {return}                                       | return | 1
          =

[Parameter]

Separator = ,{space}
Text      = {name}

; ========================================================
; Statements
; ========================================================

[Assign]
Text      = {Variable} = {Expression}

[Call]
Text      = {Name} ({Arguments})

[Comment]
Text      = # {Text}

[Declare Name]
Separator  = ,{space}
Text       = {name}

[Declare]
Name mode = Singular

Text      = {Name} = [0] * ({Size})            | Array, integer
          = {Name} = [0] * ({Size})            | Array, real
          = {Name} = [""] * ({Size})           | Array, string
          = {Name} = [False] * ({Size})        | Array, boolean
          = {Name} = [] * ({Size})             | Array, none
[Do]
Text      = while True:    #This simulates a Do Loop
          = -->BLOCK                                    || 1
          = if not({condition}): break   #Exit loop     || 1

[For]
Text      = for {Variable} in range({start}, {end} + {step}, {step}):   |  inc
          = for {Variable} in range({start}, {end} - {step}, -{step}):  |  ~inc
          = -->BLOCK                                                    |
| 1
          = pass                                                        | ~block
| 1

[Input]
Text      = {Variable} = int(input())                       | integer
          = {Variable} = float(input())                     | real
          = {Variable} = (input().lower == 'true')          | boolean
          = {Variable} = input()                            | string
          = {Variable} = input()                            | none

[If]
Text      = if {condition}:
          = -->TRUEBLOCK              |        | 1
          = pass                      | ~block | 1
```

```
          = else:                       | else
          = -->FALSEBLOCK               | else   | 1

[Output]
Text      = print({Expression})                          | newline
          = print({Expression}, end='', flush=True)      | ~newline

[While]
Text      = while {condition}:
          = -->BLOCK                    |        | 1
          = pass                        | ~block | 1
```

Linguaggio di programmazione Visual Basic .NET

```
[Language]
Name           = Visual Basic .NET
Extension      = vb

Keywords       = AddHandler, AddressOf, Alias, And, AndAlso, As, Boolean, ByRef,
Byte, ByVal
               = Call, Case, Catch, CBool, CByte, CChar, CDate, CDec, CDbl, Char
               = CInt, Class, CLng, CObj, Const, Continue, CSByte, CShort, CSng,
CStr
               = CType, CUInt, CULng, CUShort, Date, Decimal, Declare, Default,
Delegate, Dim
               = DirectCast, Do, Double, Each, Else, ElseIf, End, EndIf, Enum,
Erase
               = Error, Event, Exit, False, Finally, For, Friend, Function, Get,
GetType
               = Global, GoSub, GoTo, Handles, If, Implements, Imports, In,
Inherits, Integer
               = Interface, Is, IsNot, Let, Lib, Like, Long, Loop, Me, Mod
               = Module, MustInherit, MustOverride, MyBase, MyClass, Namespace,
Narrowing, New, Next
               = Not, Nothing, NotInheritable, NotOverridable, Object, Of, On,
Operator, Option, Optional
               = Or, OrElse, Overloads, Overridable, Overrides, ParamArray,
Partial, Private, Property, Protected
               = Public, RaiseEvent, ReadOnly, ReDim, REM, RemoveHandler,
Resume, Return, SByte, Select
               = Set, Shadows, Shared, Short, Single, Static, Step, Stop,
String, Structure
               = Sub, SyncLock, Then, Throw, To, True, Try, TryCast, TypeOf,
Variant
               = Wend, UInteger, ULong, UShort, Using, When, While, Widening,
With, WithEvents
               = WriteOnly, Xor

Conflicts      = inputText, inputValue, output

Case Sensitive = false
Options        =

; =====================================================
; Literals
; =====================================================

[Types]
Integer      = Integer
Real         = Double
Boolean      = Boolean
String       = String

[Function ID]
Convention = proper
Normal     = {Name}
Conflict   = [{Name}]

[Variable ID]
Convention = camel
Normal     = {Name}
Conflict   = [{Name}]
```

```
[String Literal]
Text           = "{Characters}"
Replace Char 1 = "
Replace By 1   = ""

[Boolean Literal]
true     =  True
false    =  False

[Integer Literal]
Text     = {Integral}

[Real Literal]
Text     = {Integral}.{Fractional}

[Variable Access]
Precedence = 100
Text       = {Name}                | ~subscript
           = {Name}({Subscript})   | subscript

; ========================================================
; Expressions
; ========================================================
; VB .NET precedence:
; 1. or
; 2. and
; 3. not
; 4. Relational and equality: =, >, < ...
; 5. Addition
; 6. Multiply
; 7. Unary: -
; 8. ^
; 100. Atom, paranthesis

[Or]
Precedence   = 1
Needed 1     = 1
Needed 2     = 2
Text         = {1} Or {2}

[And]
Precedence   = 2
Needed 1     = 2
Needed 2     = 3
Text         = {1} And {2}

;=== NOT is precedence 3

[Equals]
Precedence   = 4
Needed 1     = 5
Needed 2     = 5
Text         = {1} = {2}

[Not Equals]
Precedence   = 4
Needed 1     = 5
Needed 2     = 5
Text         = {1} <> {2}

[Less Than]
Precedence   = 4
```

```
Needed 1     = 5
Needed 2     = 5
Text         = {1} < {2}

[Less Equal Than]
Precedence   = 4
Needed 1     = 5
Needed 2     = 5
Text         = {1} <= {2}

[Greater Than]
Precedence   = 4
Needed 1     = 5
Needed 2     = 5
Text         = {1} > {2}

[Greater Equal Than]
Precedence   = 4
Needed 1     = 5
Needed 2     = 5
Text         = {1} >= {2}

[Concatenate]
Precedence   = 5
Needed 1     = 5
Needed 2     = 6
Text         = {1} & {2}

[Add]
Precedence   = 5
Needed 1     = 5
Needed 2     = 6
Text         = {1} + {2}

[Subtract]
Precedence   = 5
Needed 1     = 5
Needed 2     = 6
Text         = {1} - {2}

[Multiply]
Precedence   = 6
Needed 1     = 6
Needed 2     = 7
Text         = {1} * {2}

[Divide]
Precedence   = 6
Needed 1     = 6
Needed 2     = 7
Text         = {1} / {2}

[Modulus]
Precedence   = 6
Needed 1     = 6
Needed 2     = 7
Text         = {1} Mod {2}

[Power]
Precedence   = 8
Needed 1     = 9
Needed 2     = 8
```

```
Text         = {1} ^ {2}

[Not]
Precedence   = 3
Needed 1     = 3
Text         = Not {1}

[Negate]
Precedence   = 7
Needed 1     = 7
Text         = -{1}

[Subexpression]
Precedence   = 100
Text         = ({Expression})

; ======================================================
; Intrinsic Functions
; ======================================================

[Abs]
Precedence = 100
Needed 1   = 1
Text       = Math.Abs({1})

[ArcCos]
Precedence = 100
Needed 1   = 1
Text       = Math.ACos({1})

[ArcSin]
Precedence = 100
Needed 1   = 1
Text       = Math.ASin({1})

[ArcTan]
Precedence = 100
Needed 1   = 1
Text       = Math.Atan({1})

[Char]
Precedence = 100
Needed 1   = 100
Needed 2   = 1
Text       = {1}.CharAt({2})

[Cos]
Precedence = 100
Needed 1   = 1
Text       = Math.Cos({1})

[Int]
Precedence = 100
Needed 1   = 1
Text       = Math.Floor({1})

[Len]
Precedence = 100
Needed 1   = 100
Text       = {1}.Length

[Log]
```

```
Precedence = 100
Needed 1   = 1
Text       = Math.Log({1})

[Log10]
Precedence = 100
Needed 1   = 1
Text       = Math.Log10({1})

[Pi]
Precedence = 100
Text       = Math.PI

[Random]
Precedence = 100
Needed 1   = 1
Text       = random.Next({1})

[Sin]
Precedence = 100
Needed 1   = 1
Text       = Math.Sin({1})

[Sgn]
Precedence = 100
Needed 1   = 1
Text       = Math.Sign({1})

[Size]
Precedence = 100
Needed 1   = 100
Text       = {1}.Length

[Sqrt]
Precedence = 100
Needed 1   = 1
Text       = Math.Sqrt({1})

[Tan]
Precedence = 100
Needed 1   = 1
Text       = Math.Tan({1})

[ToChar]
Precedence = 100
Needed 1   = 1
Text       = Strings.ChrW({1})

[ToCode]
Precedence = 100
Needed 1   = 1
Text       = String.AscW({1})

[ToFixed]
Precedence = 100
Needed 1   = 1
Needed 2   = 1
Text       = ToFixed({1}, {2})

[ToInteger]
Precedence = 100
Needed 1   = 1
```

```
Text       = Convert.ToInt32({1})

[ToReal]
Precedence = 100
Needed 1   = 1
Text       = Convert.ToDouble({1})

[ToString]
Precedence = 100
Needed 1   = 100
Text       = {1}.ToString()

; ========================================================
; Function call
; ========================================================

[Function Call]
Precedence = 100
Text       = {name}({arguments})

[Argument]
Separator  = ,{space}
Text       = {expression}

; ========================================================
; Program
; ========================================================

[Program]
Text      = Imports System
          =
          = Public Module MyProgram
          = Private random as new Random()       | Random | 1
          =                                      | Random | 1

          = -->MAIN                                | | 1
          = -->FUNCTIONS                           | | 1

          =                                                                    |
tofixed | 1
          = Function ToFixed(value As Double, digits As Integer) As String     |
tofixed | 1
          = Return value.ToString("f" & digits)                                |
tofixed | 2
          = End Function                                                       |
tofixed | 1

          =
| input | 1
          = ' .NET can only read single characters or entire lines from the
console.  | input | 1
          = ' The following functions are designed to help input specific data
types. | input | 1
          = Private Function inputValue() As Double                     | input
| 1
          = Dim result As Double                                        | input
| 2
          = While Not Double.TryParse(Console.ReadLine(), result)       | input
| 2
          = ' No code in the loop                                       | input
| 3
```

```
          = End While                                                    | input
| 2
          = Return result                                                | input
| 2
          = End Function                                                 | input
| 1
          =                                                              | input
| 1
          = Private Function inputText() As String                       | input
| 1
          = return Console.ReadLine()                                    | input
| 2
          = End Function                                                 | input
| 1

          =                                                              | output
| 1
          = Private Sub output(text As String)                           | output
| 1
          = Console.WriteLine(text)                                      | output
| 2
          = End Sub                                                      | output
| 1

          = End Module

[Main]
Text      = Sub Main                      | |
          = -->BLOCK                        | | 1
          = End Sub                       | |

; ========================================================
; Function
; ========================================================

[Parameter]
Separator = ,{space}
Text      = {name} As {type}              | ~array
          = ByRef {name}() As {type}      | array

[Function]
Text      =
          = Private Sub {name} ({parameters})         | ~return
          = -->BLOCK                                  | ~return | 1
          = End Sub                                   | ~return

          = Private Function {name} ({parameters})    | return
          = -->BLOCK                                  | return | 1
          =                                           | return | 1
          = Return {return  }                         | return | 1
          = End Function                              | return

; ========================================================
; Statements
; ========================================================

[Assign]
Text      = {Variable} = {Expression}

[Call]
Text      = {Name}({Arguments})
```

```
[Comment]
Text      = ' {Text}

[Declare Name]
Separator = ,{space}
Text      = {name} As {Type}              | ~array
          = {name}({Size}) As {Type}      | array

[Declare]
Text      = Dim {Variables}

[Do]
Text      = Do
          = -->BLOCK                    | | 1
          = Loop While {condition}

[For]
Text      = For {Variable} = {Start} To {End}               | ~step
          = For {Variable} = {Start} To {End} Step {step}   | step
          = -->BLOCK                                        |       | 1
          = Next

[Input]
Text      = {Variable} = inputText()      | ~string
          = {Variable} = inputValue()     | String

[If]
Text      = If {condition} Then
          = -->TRUEBLOCK                    |       | 1
          = Else                            | else
          = -->FALSEBLOCK                   | else  | 1
          = End If

[Output]
Text      = output({Expression})

[While]
Text      = Do While {condition}
          = -->BLOCK                            | | 1
          = Loop
```