# Analisi Sintattica

Nella fase di analisi sintattica, nel tentativo di risolvere i conflitti derivanti dalla prima produzione, abbiamo identificato un problema nell'ambiguità causata dalla condivisione dello stesso stato tra ITER prima di PROCEDURE e ITER dopo.

Program -> Iter Procedure Iter

Iter -> VarDecl Iter

| Function Iter

| Procedure Iter

| ε

Questa ambiguità è stata risolta introducendo una distinzione tra ITER1 e ITER2, dove solo ITER2 può produrre una PROCEDURE.

Program -> Iter1 Procedure Iter2

Iter1 -> VarDecl Iter1

| Function Iter1

| ε

Iter2 -> VarDecl Iter2

| Function Iter2

| Procedure Iter2

| ε

Per quanto riguarda i conflitti derivanti dalla produzione:

IOArgs -> Expr IOArgs

| DOLLARSIGN LPAR Expr RPAR IOArgs

| ε

Abbiamo osservato che la ricorsione in IOARGS -> EXPR IOARGS consentiva sequenze del tipo "EXPR EXPR", portando a interpretazioni errate nel linguaggio. Abbiamo modificato la grammatica introducendo attraverso la definizione di regole più dettagliate per IOARGS.

IOArgs -> Expr

| Expr DOLLARSIGN LPAR Expr RPAR IOArgs

| DOLLARSIGN LPAR Expr RPAR IOArgs

| ε

L'analizzatore sintattico costruisce l'albero sintattico, i vari nodi dell'albero sono costituiti dalle classi presenti all'interno del package **syntaxtree**. L'albero è visitabile grazie all'implementazione dell'interfaccia **Visitor**, nel package **syntaxtree/visitor**.

# Analisi Semantica

La fase di analisi semantica per il linguaggio Toy2 viene effettuata utilizzando due visite che si occupano della gestione dello scoping e del type-checking, presenti nel package **visitor/semanticVisitor** del progetto:

- la prima, implementata con la classe **SemanticVisitorFirstVisit** crea e riempie tutte le tabelle dei simboli con le dichiarazioni di variabili, funzioni e procedure.
- la seconda, implementata con la classe **SemanticVisitorSecondVisit** effettua tutti i controlli necessari nell'analisi semantica (ad esempio variabile già dichiarata, variabile non dichiarata, etc.)

La classe **SemanticVisitorAbstract** è la superclasse dei due Visitor.

Sono state utilizzate due visite piuttosto che una in quanto le dichiarazioni di variabili, procedure e funzioni del linguaggio Toy2 possono essere effettuate in qualsiasi punto del codice (anche dopo il loro utilizzo).

## Gestione dello Scoping

Per la gestione dello Scoping viene utilizzata la classe **SymbolTable**, nel package **table** che espone i seguenti metodi:

- **enterScope**: crea una nuova tabella (**SymbolNode**) figlia di quella correntemente attiva che va a sostituire quest'ultima;
- **enterSpecificScope**: utilizzata nella seconda visita, permette di entrare nello scope specifico;
- **addChildToParentScope**: nella prima visita aggiunge la tabella come figlia della tabella padre per gli utilizzi nella seconda visita;
- **exitScope**: rimuove la tabella attualmente attiva e la sostituisce con il padre;
- **lookup**: prende in input un id, risale l'albero delle tabelle a partire da quella attualmente attiva e restituisce il primo elemento associato a quell'id che trova, nel caso non trovi tale elemento restituisce null;
- **lookupOnlyActive**: come lookup ma cerca solo nella tabella corrente;
- **probe**: prende in input un id e restituisce true se questo è presente nello scoping attuale;
- **addId**: aggiunge un nuovo id alla tabella di scoping attiva.

Un nuovo scope viene creato ai seguenti punti:

- sul nodo **ProgramOp**,
- sul nodo **FunDeclOp**,
- sul nodo **ProcOp**
- sui nodi **BodyOp** associati ai costrutti if, else if, else e while.

Invece si esce dallo scoping corrente alla fine delle operazioni di visita sul nodo **BodyOp**.

I nodi che invece provvedono a inserire un nuovo id nella tablella corrente sono **FunDeclOp**, **ProcOp** e **VarDeclOp**, che oltre all'id inseriscono nella tabella anche le informazioni sul tipo, per quanto riguarda le dichiarazioni di variabili, sul tipo di ritorno per le dichiarazioni di funzioni e sul tipo dei parametri per la dichiarazione di funzioni e procedure.

Nella prima visista viene effettuata anche l'inferenza di tipo per le variabili dichiarate direttamente con il loro valore, le quali assumeranno il tipo della costante con cui sono state inizializzate. L'inserimento dell'id nella tabella può essere effettuato solo se questo non è già presente nello scoping attuale, in caso contrario viene lanciata un'eccezione.

Ogni volta che invece un id viene utilizzato viene controllato con la funzione lookup, che questo sia stato in precedenza dichiarato.

Durante questa fase viene attribuito un tipo ai nodi espressione e viene controllato che i vari costrutti rispecchino il Type-System del linguaggio Toy2.

Oltre alle regole di compatibilità presenti nelle specifiche di Toy2, per l'assegnazione e per i parametri attuali e formali si accettano soltanto tipi uguali tranne per interi e reali per i quali è possibile anche assegnare un intero a un reale.

## Errori

Gli errori possibili nella fase dell'analisi semantica sono:

- variabile già dichiarata nello scope corrente;
- chiamata di procedure e funzioni non dichiarate;
- variabile non dichiarata né nello scope corrente che in quelli padre;
- incompatibilità dei tipi dei parametri, tipi di ritorno e variabili;
- passaggio di un parametro OUT non in riferimento e viceversa;
- numero di parametri dichiarati corrispondente al numero di parametri passati in funzioni e procedure;
- numero di valori di ritorno dichiarati corrispondente al numero di valori di ritorno passati nelle funzioni;
- procedura main non dichiarata;
- return non previsto nella procedura;
- mancanza del return in una funzione;
- controllo che le espressioni dei costrutti if, else if e while siano di tipo booleano;
- le chiamate a funzioni che ritornano più di un valore possono essere usate solo come parte destra delle assegnazioni;
- numero di parametri a destra non corrispondente alle espressioni di sinistra nell'assegnazione;
- passaggio di un valore che non è un id di variabile in $() in READ e dopo @ nelle chiamate a procedura;

# Traduzione in C

La traduzione in linguaggio C viene effettuata con la classe **CVisitor** nel package **visitor**, che esegue un'ulteriore visita sull'AST arricchito con le informazioni aggiunte durante l'analisi semantica.

## Gestione delle stringhe

Il tipo string del linguaggio Toy2 viene sostituito dal tipo char in C, in particolare tutte le stringhe hanno dimensione fissa (default 256 caratteri). Ad esempio, se la stringa non è inizializzata:

```
char *s = (char *)malloc(256 * sizeof(char));
```

se è stata inizializzata, si aggiunge la funzione strcpy usata anche quando viene assegnato un valore ad una stringa:

```
char *s = (char *)malloc(256 * sizeof(char));
strcpy(s, "prova");
```

Per la concatenazione viene utilizzata la funzione:

```
char *myStrcat(char *s1, const char *s2){
    char *s = (char *)calloc(strlen(s1) + strlen(s2) + 1, sizeof(char));
    strcat(s, s1);
    strcat(s, s2);
    return s;
}
```

Per la conversione da intero a stringa e da reale a stringa vengono usate le seguenti funzioni:

```
char *intToString(int num){
    char *buffer = (char *)malloc(30 * sizeof(char));
    snprintf(buffer, 30, "%d", num);
    return buffer;
}
char *floatToString(float num){
    char *buffer = (char *)malloc(30 * sizeof(char));
    snprintf(buffer, 30, "%f", num);
    return buffer;
}
```

Per il confronto tra due stringhe viene usata la funzione:

```
strcmp(s1, s2) != 0    //per verificare che sono diverse in NeOp
strcmp(s1, s2) == 0    //per verificare che sono uguali in EqOp
```

## Parametri di tipo OUT

Per i parametri OUT nelle procedure sono stati utilizzati i puntatori in C. Tranne per le stringhe che sono sempre passate per indirizzo, la traduzione della dichiarazione di un parametro in modalità OUT è:

`out x: integer` diventa `int *x`

Per quanto riguarda il passaggio di un parametro OUT in una chiamata a funzione, viene passato l'indirizzo della variabile, ad esempio:

`@x` diventa `& x`

## Istruzioni READ e WRITE

L'istruzione di read viene tradotta nel seguente modo:

```
<-- "prova" $(x); diventa
    printf("prova");
    scanf("%d", &x);
```

L'istruzione write:

```
--> "prova" $(x); diventa
 printf("prova %d", x);
```

L'istruzione write retur

```
n:-->! "prova" $(x); diventa
 printf("prova %d\n", x);
```

| modificatore | tipo |
|---|---|
| %s | string |
| %d | integer, boolean |
| %f | real |

## Funzioni che restituiscono più di un valore di ritorno

Le funzioni che restituiscono più di un valore di ritorno (che possono essere usate solo nell'assegnazione) vengono trasformate in funzioni void e vengono aggiunti come parametri un numero di parametri puntatore con lo stesso tipo, pari al numero di valori che la funzione dovrebbe restituire. Queste variabili aggiuntive vengono dichiarate prima della chiamata di funzione, passate per riferimento come parametri e successivamente assegnate alle rispettive variabili. Ad esempio:

```
func func1(x : integer) -> interger, integer:
      return 3 , 3;
endfunc;
```

diventa

```
void func1(int x, int* p0, int *p1){
      p0=3;
      p1=3;
}
```

Per le chiamate di funzione, invece:

```
a, b ^= func(x);
```

diventa

```
int p0;
int p1;
func(x, &p0, &p1);
a = p0;
b = p1;
```