



Politecnico di Bari

Corso:

Formal Language and Compilers

Transpilatore di un sub-set Lua a C

Docente: Floriano Scioscia

Autori:

Francesco Faienza matricola: 596856

Email: f.faienza@studenti.poliba.it

Simone Tavilla matricola: 598066

Email: s.tavilla@studenti.poliba.it

Indice

1	Introduzione	1
1.1	Sub-set Lua	1
1.2	Struttura del Progetto	2
2	Analisi Lessicale	3
2.1	Opzioni e Variabili di Supporto	3
2.2	Gestione degli Errori e <code>copy_line()</code>	4
2.3	Attributi dei Token	4
2.4	Riconoscimento di Commenti e Stringhe mediante Stati	5
2.5	Riconoscimento di Numeri, Parole Chiave e Identificatori	6
2.5.1	Costanti Numeriche	6
2.5.2	Parole Chiave e Identificatori	6
2.6	Operatori e Altri Simboli	7
2.7	Gestione degli Spazi Bianchi e Fallback per Errori	7
3	Analisi Sintattica	8
3.1	Gestione degli Scope	8
3.2	Definizione di funzione	9
3.2.1	Definizioni Funzione Standard	10
3.2.2	Definizione di Funzione Assegnata e Gestione Sintattica di <code>io.read</code>	10
3.2.3	Gestione Sintattica di <code>print</code>	11
4	AST e SymTable	12
4.1	Abstract Syntax Tree	12
4.1.1	Struttura del Nodo	12
4.2	Symbolic Table	12
4.2.1	Struttura tabelle	13
4.2.2	Scope	13
5	Semantica	14
5.1	Inferenza dei Tipi nel Transpiler	14
5.2	Gestione Semantica di <code>io.read</code>	16
5.3	Inferenza del Tipo di Ritorno delle Funzioni	17
6	Traduzione	19
6.1	Test	20

1 Introduzione

L'obiettivo è stato realizzare un transpilatore[5] che permetta di tradurre un sottoinsieme di istruzioni del linguaggio **Lua**[4] al linguaggio **C**[2].

Sono stati scelti questi linguaggi per dimostrare un caso d'uso reale di un transpilatore di questo tipo. Infatti la traduzione di un linguaggio dinamico e interpretato come Lua può avere molti vantaggi nell'essere tradotto in un linguaggio tipizzato e compilato come C, che vanno dalla velocità di esecuzione ai benefit di un programma tipizzato.

Il transpilatore è stato realizzato in linguaggio C con l'ausilio del generatore di scanner automatico **Flex** (Fast Lexical Analyzer Generator) per l'analisi del lessico e il generatore di parser **GNU Bison** per l'analisi semantica. Inoltre è stata utilizzata la libreria **uthash** per l'implementazione della symbol table, gestita come hash table.

1.1 Sub-set Lua

La restrizione del linguaggio sorgente Lua prevede le seguenti istruzioni da tradurre nel linguaggio di arrivo:

- **Tipi di dato:** number (sia interi che floating), string, bool;
- **Operatori aritmetici:** somma, sottrazione, moltiplicazione, divisione;
- **Operatori logici:** and, or, not;
- **Struttura dati:** table (struttura dati simile a un dizionario);
- **Operatori di confronto:** minore, minore o uguale, uguale, maggiore, maggiore o uguale, diverso;
- **Commenti:** commenti `--COMMENTO` e `--[COMMENTO]` -
- **Istruzioni di jump:** if, else;
- **Istruzione di iterazione:** for;
- **Istruzione di input:** `io.read` (standard library I/O);
- **Istruzione di output:** `print` (funzione built-in).

Oltre le precedenti istruzioni, il transpilatore realizzato supporta le seguenti funzionalità:

- **Dichiarazione e assegnazione di variabili** rispettando lo scope di appartenenza;
- **Definizione e chiamata di funzioni:** con l'eccezione di avere una sola variabile di ritorno a causa di restrizioni date dal linguaggio C;
- **Espressioni:** fanno uso di operatori logici, aritmetici e di confronto e prevedono la possibilità di utilizzare variabili.

1.2 Struttura del Progetto

Il progetto è organizzato nei seguenti file:

- **global.h** è il file contenente le variabili globali e i prototipi delle funzioni utilizzate in più file (ad esempio la definizione delle costanti per i colori dei testi e i prototipi delle funzioni per la creazione e la presentazione dei messaggi di errore);
- **scanner.l** è il file contenete le specifiche per la generazione dello scanner mediante il tool Flex. In aggiunta, contiene le definizioni delle funzioni che permettono di gestire la creazione e la presentazione dei messaggi di errore;
- **parser.y** è il file contenente le specifiche necessarie a generare il parser mediante il tool Bison. In aggiunta, contiene il main e le funzioni di supporto alla creazione e chiusura degli scope;
- **ast.h** e **ast.c** sono i file contenenti la struttura dei nodi dell'Abstract Syntax Tree e le funzioni che ne permettono la creazione e la modifica dei loro attributi;
- **pretty.c** e **pretty.h** sono i file contenenti le funzioni per stampare su stout il parse tree generato;
- **syntab.h** e **syntab.c** sono i file contenti la struttura delle Symbol Table e le funzioni che permettono la gestione delle tabelle e dei loro simboli. Inoltre, contengono le strutture e le funzioni per gestire la gerarchia delle tabelle in accordo con gli scope;
- **semantic.h** e **semantic.c** sono i file contenti le funzioni relative ai controlli semantici implementati dal compilatore;
- **translate.h** e **translate.c** sono i file contenti le funzioni utilizzate durante la traduzione;
- **uthash.h** è una libreria usate per implementare hash table in C. È utilizzata come supporto per la gestione delle Symbol Table.

Per ciascuna tipologia di istruzione da tradurre sono inoltre presenti due file: il primo contenente codice Lua scritto correttamente mentre il secondo contiene codice volutamente errato in modo da valutare la gestione degli errori da parte del transpilatore. Il codice C di output viene stampato nei file `<file>.c` e `<file>.h` mentre la stampa degli errori sintattici e semantici viene effettuata sullo stdout.

2 Analisi Lessicale

La prima fase di un compilatore, o nel nostro caso di un transpiler, è l'**analisi lessicale**. In questa fase, il codice sorgente in input (Lua) viene letto carattere per carattere e raggruppato in sequenze significative chiamate **lessemi**. Ad ogni lessema riconosciuto corrisponde un **token**, che ne rappresenta la categoria sintattica (ad esempio, identificatore, parola chiave, operatore, costante numerica). Il componente che esegue questa operazione è detto **scanner** o analizzatore lessicale.

Per la generazione dello scanner del nostro transpiler Lua-to-C, è stato utilizzato lo strumento **Flex** (Fast LEXical analyzer generator)[3]. Flex è un'alternativa software gratuita e più performante a LEX, ampiamente utilizzata in combinazione con **Bison** (un generatore di parser alternativo a YACC)[1]. Flex prende in input un file di specifiche (solitamente con estensione .l) contenente definizioni di token sotto forma di espressioni regolari e le azioni C associate da eseguire al riconoscimento di ciascun token. L'output di Flex è un file sorgente C (`lex.yy.c`) che implementa la funzione `yylex()`, la quale costituisce il cuore dello scanner.

Il file di input per Flex, nel nostro caso `scanner.l`, è strutturato in tre sezioni separate da `%`:

1. **Sezione Definizioni:** Contiene direttive per Flex (opzioni), inclusioni di file header C, definizioni di stati e macro utili per le espressioni regolari.
2. **Sezione Regole:** Costituisce il nucleo dello scanner, definendo i pattern (espressioni regolari) da riconoscere e le corrispondenti azioni C da eseguire.
3. **Sezione Codice Utente:** Contiene codice C ausiliario, come funzioni di supporto richiamate nelle azioni della sezione regole.

Di seguito, verranno analizzate le parti più significative del file `scanner.l` del nostro progetto.

2.1 Opzioni e Variabili di Supporto

```
%option yylineno
```

L'opzione `%option yylineno` abilita in Flex il tracciamento automatico del numero di riga corrente del file di input. Il numero di riga è memorizzato nella variabile globale `yylineno` e consente di fornire messaggi di errore dettagliati con indicazione precisa della posizione nel file sorgente.

```
char string_buf[230000];
```

Il buffer `string_buf` viene utilizzato per accumulare i caratteri delle stringhe letterali riconosciute dallo scanner. La sua dimensione è stata scelta per garantire la gestione di stringhe in Lua della massima lunghezza offerta dal linguaggio.

2.2 Gestione degli Errori e `copy_line()`

Per migliorare la diagnostica degli errori, è stata implementata una strategia che permette di visualizzare la riga del codice sorgente in cui si è verificato un errore. Questo è possibile grazie alla funzione `copy_line()` e alla regola associata.

```
^.* { copy_line(); }
```

La regola `^.*` corrisponde a qualsiasi sequenza di caratteri dall'inizio (^) della riga fino alla sua fine. Quando questa regola viene attivata, viene chiamata la funzione `copy_line()`.

```
void copy_line() {
    if(line) {
        free(line);
    }

    line = malloc(sizeof(char) * (yyleng + 1));
    strcpy(line, yytext);

    yyless(0);
}
```

La funzione `copy_line()` viene usata per salvare la riga corrente del file che lo scanner sta leggendo. Quando Flex legge una riga, quella riga è contenuta nella variabile `yytext`. La lunghezza della riga è `yyleng`. Quello che fa `copy_line()` è:

- se c'è già una riga salvata, la cancella (libera la memoria);
- copia la riga `yytext` nella variabile globale `line`;
- poi rimette la riga nel buffer di input con `yyless(0)`, così le regole di Flex possono leggerla di nuovo, come se non fosse mai stata letta.

Questo meccanismo assicura che, dopo aver salvato la riga per eventuali messaggi di errore, i caratteri vengano riprocessati dalle regole successive dello scanner per il corretto riconoscimento dei token.

I messaggi di errore sono gestiti dalla funzione `yyerror()`, che stampa un messaggio formattato sullo standard error, includendo il nome del file, il numero di riga (`yylineno`) e la riga stessa (`line`).

```
void yyerror(const char *s) {
    fprintf(stderr, "%s:%d " RED "error:" RESET " %s\n", filename, yylineno, s);
    fprintf(stderr, "%s\n", line);
    error_num++;
}
```

2.3 Attributi dei Token

Durante l'analisi lessicale, è spesso necessario associare ai token riconosciuti informazioni aggiuntive, dette *attributi*. Ad esempio, per identificatori, numeri e stringhe, è utile conservare il lessema corrispondente.

```
yylval.s = strdup(yytext);
```

In Flex, il lessema riconosciuto è disponibile nella variabile globale `yytext`, di tipo `char*`. Poiché il contenuto di `yytext` viene sovrascritto a ogni nuovo riconoscimento, è necessario, quindi, copiarne il valore quando si desidera conservarlo. A tal fine, è stata utilizzata la funzione `strdup()` per duplicare il lessema e assegnarlo a un campo appropriato della variabile globale `yyval`, definita come un'unione (`union`) nel file del parser.

2.4 Riconoscimento di Commenti e Stringhe mediante Stati

Oltre allo stato `INITIAL`, definito di default da Flex, vengono utilizzati tre stati esclusivi:

- `COMMENT` per il riconoscimento dei commenti multi-linea,
- `DQUOTE` per le stringhe delimitate da doppi apici,
- `SQUOTE` per le stringhe delimitate da apici singoli.

L'uso degli stati consente di isolare il riconoscimento di questi costrutti complessi e di rilevare eventuali errori di terminazione.

Commenti Multi-linea

"--[["	{ BEGIN COMMENT; }
<COMMENT>[^\]]*	{ }
<COMMENT>\][^\]]*	{ }
<COMMENT>"]]"	{ BEGIN INITIAL; }
<COMMENT><<EOF>>	{ yyerror("unterminated comment"); BEGIN INITIAL; }

Lo stato `COMMENT` è attivato alla lettura di `--[[` e viene disattivato al riconoscimento della sequenza di chiusura `]]`. Se la fine del file viene raggiunta prima della chiusura, viene segnalato un errore.

Stringhe con Doppi Apici

\"	{ BEGIN DQUOTE; string_buf[0] = '\0'; }
<DQUOTE>([^\\"\\n] \\.)+	{ strcat(string_buf, yytext); }
<DQUOTE>\"	{ BEGIN INITIAL; yyval.s = strdup(string_buf); return STRING; }
<DQUOTE>\n <<EOF>>	{ yyerror("missing terminating \" character"); BEGIN INITIAL; }

L'apertura con `"` attiva lo stato `DQUOTE`, durante il quale i caratteri della stringa vengono accumulati in `string_buf`. La chiusura corretta ripristina lo stato iniziale e restituisce il token `STRING`, mentre `newline` o `EOF` producono un errore.

Stringhe con Apici Singoli

```

\'          { BEGIN SQUOTE; string_buf[0] = '\0'; }
<SQUOTE>([~'\n]|\\.)+ { strcat(string_buf, yytext); }
<SQUOTE>\'   { BEGIN INITIAL; yylval.s = strdup(string_buf); return
    STRING; }
<SQUOTE>\n | <<EOF>> { yyerror("missing terminating ' character"); BEGIN INITIAL;
    }

```

Il funzionamento è analogo allo stato DQUOTE, ma applicato alle stringhe delimitate da apici singoli.

2.5 Riconoscimento di Numeri, Parole Chiave e Identificatori

2.5.1 Costanti Numeriche

Lo scanner riconosce costanti intere e in virgola mobile.

```

/* costanti numeriche */

[0]+ |
[1-9][0-9]*          { yylval.s = strdup(yytext); return INT_NUM; }

[0]+[0-9]+          { yyerror("octal literal not allowed"); }

([0-9]+)?(\.[0-9]+) |
([0-9]+)\. |
([0-9]+)(e|E)(\+|-)?[0-9]+ |
([0-9]+)?(\.[0-9]+)(e|E)(\+|-)?[0-9]+ |
((([0-9]+)\.)(e|E)(\+|-)?[0-9]+) { yylval.s = strdup(yytext); return FLOAT_NUM; }

```

Le regole per i numeri interi sono `[0]+` (per lo zero) e `[1-9][0-9]*` (per interi positivi maggiori di zero). Viene anche inclusa una regola per identificare e segnalare un errore per i letterali ottali (es. `0123`), che non sono standard in Lua. Le espressioni regolari per i numeri in virgola mobile (FLOAT_NUM) coprono vari formati, inclusa la notazione scientifica (es. `1.23`, `.5`, `10.`, `1e-5`, `3.14E+2`). Per entrambi i tipi di numeri, il lessema viene memorizzato come stringa in `yylval.s`.

2.5.2 Parole Chiave e Identificatori

Le parole chiave del linguaggio Lua sono definite con regole specifiche.

```

/* keyword */

"and"          { return AND; }
"do"           { return DO; }
/* ... altre keyword ... */
"false"        { yylval.s = strdup(yytext); return BOOL; }
"true"         { yylval.s = strdup(yytext); return BOOL; }
"nil"          { return NIL; }

/* identificatori */

[_a-zA-Z][_a-zA-Z0-9]* { yylval.s = strdup(yytext); return ID; }

```


Come mostrato, ogni parola chiave (es. "and", "do", "if") ha una regola che restituisce il token corrispondente (es. AND, DO, IF), definito in `parser.tab.h`. Per le costanti booleane `true` e `false`, e per `nil`, oltre al token (BOOL, NIL), viene anche salvato il lessema.

La regola per gli identificatori, `[_a-zA-Z][_a-zA-Z0-9]*`, riconosce sequenze che iniziano con una lettera o un underscore, seguite da zero o più lettere, numeri o underscore. È importante che le regole per le parole chiave precedano quella per gli identificatori. Questo perché Flex applica la "regola del match più lungo" (longest match) e, in caso di parità, la regola che appare per prima nel file `.l`. Se la regola per gli identificatori precedesse quelle delle parole chiave, una keyword come "if" verrebbe erroneamente riconosciuta come un identificatore.

2.6 Operatori e Altri Simboli

Lo scanner riconosce operatori aritmetici, di confronto e altri simboli.

```
/* operatori aritmetici */
"+" | "-" | "*" | "/" | "=" { return yytext[0]; }

/* operatori di confronto */
">" | "<"           { return yytext[0]; }
">="              { return GE; }
"<="              { return LE; }
"=="              { return EQ; }
"!="              { return NE; } /* Operatore di disuguaglianza in Lua */

/* altri simboli */
"{" | "}" | "(" | ")" | "[" | "]" | ";" | "," | "&" { return yytext[0]; }
"."          { return DOT; }
```

Per gli operatori e i simboli composti da un singolo carattere (es. +, =, ()), l'azione `return yytext[0];` restituisce il carattere stesso come token. Per operatori composti da più caratteri (es. >=, ==, =), vengono restituiti token specifici (GE, EQ, NE). Anche qui, l'ordine è importante: >= deve precedere > per garantire il corretto riconoscimento.

2.7 Gestione degli Spazi Bianchi e Fallback per Errori

Gli spazi bianchi, le tabulazioni e i newline sono generalmente ignorati.

```
/* spazi e tabulazioni */
[ \t\v\f\n]          { } /* Ignora */

/* fallback per gli errori */
.                    { yyerror(error_string_format("invalid character"
    BOLD " %s" RESET, yytext)); }
```

La regola `[\t\v\f\n] { }` fa sì che questi caratteri vengano consumati senza generare alcun token. Infine, la regola `.` (punto) funge da "fallback": corrisponde a qualsiasi carattere non riconosciuto da nessuna delle regole precedenti. In questo caso, viene invocata la funzione `yyerror()` per segnalare la presenza di un carattere non valido. Questa regola dovrebbe essere l'ultima tra quelle che producono azioni, per catturare tutto ciò che non è stato specificamente definito.

3 Analisi Sintattica

L'analisi sintattica, nota anche come parsing, rappresenta la seconda fase del processo di traduzione. Il suo obiettivo primario è determinare se la sequenza di token, fornita dall'analizzatore lessicale (nel nostro caso, generato da Flex), costituisce una frase sintatticamente valida secondo le regole del linguaggio sorgente. La sintassi di un linguaggio di programmazione è comunemente descritta attraverso una grammatica libera dal contesto (Context-Free Grammar, CFG di Tipo 2), in cui ogni token è trattato come un simbolo atomico.

Il parser, quindi, opera su questa sequenza di token tentando di costruire una struttura gerarchica, nota come parse tree, che dimostri come la sequenza di token possa essere derivata dalle regole grammaticali. Se tale albero può essere costruito, la sequenza è considerata sintatticamente corretta e l'albero stesso (nel nostro caso una sua forma astratta l'Abstract Syntax Tree - AST) viene passato alle fasi successive della compilazione. In caso contrario, il parser segnala un errore sintattico.

Nel contesto del nostro transpiler da Lua a C, abbiamo impiegato lo strumento Bison per generare l'analizzatore sintattico. Il file `parser.y` costituisce l'input per Bison, contenente la definizione formale della grammatica del linguaggio Lua (o meglio della sua porzione rilevante per il nostro transpiler) sotto forma di regole di produzione, e le azioni semantiche in C associate a tali regole, che verranno eseguite al riconoscimento di specifici costrutti sintattici.

Questo capitolo si presuppone di fornire una panoramica della struttura e delle componenti significative del nostro file `parser.y`.

3.1 Gestione degli Scope

```
program: { scope_enter(); } global_statement_list { root = $2; scope_exit(); };
```

All'inizio dell'analisi del programma, viene aperto lo scope globale. Alla fine, viene chiuso.

```
void scope_enter() {
    current_syntab = create_syntab(current_scope_lvl, current_syntab);

    if (root_syntab == NULL) {
        root_syntab = current_syntab;
    }
    current_scope_lvl++;

    if(param_list) {
        fill_syntab(current_syntab, param_list, -1, PARAMETER);
        param_list = NULL;
    }

    ret_type = -1; // Resetta ret_type al valore di default
}
```

La funzione `scope_enter()` viene chiamata ogni volta che il parser entra in un costrutto che definisce un nuovo scope (ad esempio: l'inizio di una funzione, il blocco `then` di un `if`, il corpo di un `for`).

- Crea una nuova tabella dei simboli: utilizza `create_syntab(current_scope_lvl, current_syntab)`. La nuova tabella viene collegata alla precedente (che diventa il suo genitore, accessibile tramite il campo `next` della nuova tabella). `current_syntab` viene aggiornato per puntare a questa nuova tabella.
- Inizializza `root_syntab`: se è il primo scope ad essere creato (quindi `root_syntab == NULL`), `root_syntab` viene impostato a `current_syntab`.
- Incrementa `current_scope_lvl`: segnala l'ingresso in un livello di scope più profondo.
- Gestisce i parametri di funzione: se la variabile globale `param_list` (un `struct AstNode*` che punta a una lista di nodi AST rappresentanti i parametri) è stata popolata (tipicamente durante l'analisi di una definizione di funzione), `scope_enter()` utilizza `fill_syntab()` per inserire questi parametri nella tabella dei simboli appena creata (quella dello scope della funzione). Dopodiché, `param_list` viene resettata a `NULL`.
- Resetta `ret_type`: una variabile globale `ret_type` (usata per il tipo di ritorno delle funzioni) viene resettata, preparandola per un'eventuale nuova inferenza all'interno del nuovo scope.

```
void scope_exit() {
    if(print_syntab_flag)
        print_syntab(current_syntab);

    current_syntab = current_syntab->next;
    current_scope_lvl--;
}
```

La funzione `scope_exit()` viene chiamata quando il parser esce da un blocco che definiva uno scope.

- Stampa opzionale della tabella dei simboli: se il flag `print_syntab_flag` è impostato, la tabella dei simboli dello scope che si sta chiudendo (`current_syntab`) viene stampata.
- Ripristina lo scope genitore: `current_syntab` viene aggiornato al suo campo `next` (cioè `current_syntab = current_syntab->next;`), facendo tornare attivo lo scope genitore.
- Decrementa `current_scope_lvl`: segnala l'uscita dallo scope corrente e il ritorno al livello precedente.

3.2 Definizione di funzione

La regola `func_definition` nel file `parser.y` è responsabile del riconoscimento delle diverse forme sintattiche con cui una funzione può essere definita in LUA.

3.2.1 Definizioni Funzione Standard

Le produzioni principali per le definizioni di funzioni nominali (con nome eslicito) sono:

```
func_definition
: FUNCTION ID '(' param_list ')' chunk END
| FUNCTION ID '(' ' ' ')' chunk END
;
```

- **FUNCTION:** la definizione inizia sempre con la parola chiave **FUNCTION**.
- **ID:** segue un identificatore, che rappresenta il nome della funzione. Questo identificatore viene catturato dal parser come token **ID**.
- **Parentesi e Parametri:**
 - **'(' param_list ')':** se la funzione accetta parametri, questi sono racchiusi tra parentesi tonde. La sottoregola **param_list** analizza la lista di uno o più parametri, che sono a loro volta **ID** o altre espressioni valide come parametri (anche se nel caso attuale si concentra su **ID** e valori letterali per default). Il risultato di **param_list** è un nodo AST che struttura questi parametri.
 - **'(' ' ')':** se la funzione non ha parametri, vengono riconosciute semplicemente le parentesi vuote.
- **chunk:** rappresenta il corpo della funzione, ovvero la sequenza di statement **LUA** che verranno eseguiti quando la funzione è chiamata. Il non terminale **chunk** si espande ulteriormente in **statement_list**.
- **END:** la definizione della funzione termina con la parola chiave **END**.

3.2.2 Definizione di Funzione Assegnata e Gestione Sintattica di `io.read`

Una forma sintattica cruciale, soprattutto per come `io.read` è gestita nel progetto, è la definizione di una funzione anonima che viene immediatamente assegnata a un nome (che può essere un semplice **ID** o, nel caso specifico, `io.read`).

```
| name_or_ioread '=' FUNCTION '(' param_list ')' chunk END
```

`name_or_ioread` è il punto chiave per `io.read`. Questa regola è definita come:

```
name_or_ioread
: ID
| ID DOT ID
;
```

La produzione `ID DOT ID` è quella responsabile del riconoscimento della sintassi `io.read`. Quando il parser incontra la sequenza di token **ID** (con valore “io”), **DOT** (il punto) e **ID** (con valore “read”), la produzione `name_or_ioread` viene soddisfatta. L’azione semantica associata a tale produzione è:

```
{ $$ = new_io_read_identififier_node($1, $3); }
```

La funzione `new_io_read_identifier_node`, definita nella sezione C del file `parser.y`, svolge un ruolo cruciale. Essa verifica che i due identificatori siano precisamente “io” e “read”. In tal caso, costruisce un nodo dell’Abstract Syntax Tree (AST) di tipo `VAR_T`, il cui nome è la stringa letterale “io.read”. Questo implica che, a livello di AST, l’accesso `io.read` viene trattato come un singolo identificatore di variabile, sebbene semanticamente rappresenti una funzione. Se invece i due identificatori non corrispondono esattamente a “io” e “read”, viene generato un errore, poiché solo `io.read` è supportato dal parser per questo tipo di accesso diretto.

Il nodo `VAR_T("io.read")` risultante costituisce il valore semantico (`$$`) della produzione `name_or_ioread`. Successivamente, il parser si attende il token di assegnazione “=” e quindi la sequenza `FUNCTION '(' param_list ')'` chunk `END`, che corrisponde alla definizione di una funzione anonima. Tale parte è strutturalmente identica alla definizione standard di una funzione senza nome esplicito.

3.2.3 Gestione Sintattica di `print`

A differenza di `io.read`, la funzione `print` non riceve un trattamento sintattico speciale o dedicato nel file `parser.y`. Il parser tratta `print(...)` come la chiamata a una qualsiasi altra funzione identificata da un identificatore (ID).

La distinzione che `print` sia una funzione predefinita (built-in) non emerge a questo livello dell’analisi sintattica, ma viene gestita successivamente nella fase semantica e durante la traduzione, dove sarà interpretato correttamente il significato dell’identificatore.

4 AST e SymTable

4.1 Abstract Syntax Tree

Durante la fase di parsing in `parser.y` è effettuata la creazione dell'Abstract Syntax Tree, una rappresentazione dei costrutti della grammatica sotto forma di albero. Questa struttura dati è indispensabile per la fase di analisi semantica e per la traduzione.

4.1.1 Struttura del Nodo

La struttura del nodo è unica, chiamata `AstNode`, e la sua struttura è definita in tabella 4.1 e comprende:

AstNode		
nodetype	node	next

Tabella 4.1: Struttura dell'AstNode

- **nodetype**: il tipo di nodo dal quale dipendono le strutture valorizzate all'interno della union `node`. I tipi possibili sono `11(EXPR_T, VAL_T, VAR_T, TABLE_FIELD_T, TABLE_NODE_T, DECL_T, RETURN_T, FCALL_T, FDEF_T, IF_T, FOR_T, ERROR_NODE_T)`;
- **node**: è una `union` che contiene diverse strutture relative a differenti tipi di nodo. I puntatori ai nodi figli del nodo corrente sono contenuti all'interno di tali strutture;
- **next**: puntatore al nodo fratello del nodo corrente e viene instaurato il collegamento tramite la funzione `link_AstNode`.

variable	
<i>name</i>	<i>table_key</i>

returnNode
<i>expr</i>

expression		
<i>expr_type</i>	<i>l</i>	<i>r</i>

value		
<i>val_type</i>	<i>string_val</i>	

funcDef		
<i>name</i>	<i>params</i>	<i>code</i>

declaration	
<i>var</i>	<i>expression</i>

ifNode		
<i>code</i>	<i>body</i>	<i>else_body</i>

forNode		
<i>varname</i>	<i>start</i>	<i>end</i>

tableField	
<i>key</i>	<i>value</i>

table
<i>fields</i>

funcCall			
<i>name</i>	<i>args</i>	<i>func_expr</i>	<i>return_type</i>

4.2 Symbolic Table

La Symbol Table è una struttura dati usata dal compilatore per tenere traccia della semantica degli identificatori. La struttura scelta per la Symbol Table è una hash table per la sua efficienza e gestione semplice del codice. Per la gestione della hash table si è scelto di utilizzare la libreria di supporto `uthash`.

4.2.1 Struttura tabelle

Ogni symbol table contiene un record per ogni identificatore, con dei campi per gli attributi. In particolare i record vengono creati durante la fase di parsing e non durante la fase di scan. La struttura dei record è rappresentata dalla `struct symbol`, i cui campi [4.2] sono:

<i>name</i>	<i>type</i>	<i>sym_type</i>	<i>pl</i>	<i>used_flag</i>	<i>lineno</i>	<i>line</i>
-------------	-------------	-----------------	-----------	------------------	---------------	-------------

Tabella 4.2: Struttura dei record nella symbol table

- **name**: rappresenta il nome del simbolo, è utilizzato come chiave per la hashtable;
- **type**: rappresenta il tipo delle variabili;
- **sym_type**: è il tipo di simbolo. I tipi sono `VARIABLE`, `FUNCTION_SYM`, `PARAMETER`, `F_RETURN`, utilizzati rispettivamente per variabili, funzioni, parametri e il simbolo speciale per return. Questo simbolo viene inserito all'interno della tabella relativa allo scope di una funzione.
- **pl**: è un puntatore alla lista dei parametri della funzione;
- **used_flag**: è un flag che indica se una variabile è già stata dichiarata, utile in fase di traduzione per distinguere il caso di dichiarazione e assegnazione, interpretato allo stesso modo in Lua;
- **lineno**: è utilizzato per salvare il numero di riga in cui il simbolo viene dichiarato, in modo da poter presentare errori più esaurienti;
- **line**: è utilizzato per salvare una copia della riga in cui il simbolo viene dichiarato, in modo da poter presentare errori più esaurienti.

4.2.2 Scope

La gestione degli scope è stata discussa nel capitolo precedente; è opportuno descrivere i campi della tabella creata per il relativo scope. La gerarchia di tali tabelle è gestita mediante uno stack, implementato come una lista linkata. La struttura della lista consiste di una `struct symlist`, formata da [4.3]:

<i>scope</i>	<i>symtab</i>	<i>next</i>
--------------	---------------	-------------

Tabella 4.3: Struttura symlist

- **scope**: rappresenta il numero dello scope;
- **symtab**: un puntatore alla tabella corrente;
- **next**: un puntatore alla tabella successiva nello stack, ovvero lo scope più esterno.

5 Semantica

Le fasi precedenti di analisi lessicale (*scanning*) e sintattica (*parsing*) hanno permesso di trasformare il codice sorgente da una sequenza di caratteri a una sequenza di token e, successivamente, di verificare se tale sequenza di token rispetta la grammatica formale del linguaggio, producendo, nel nostro caso, un *Abstract Syntax Tree* (AST). Finora abbiamo visto come un parser identifica le derivazioni di una sequenza di token, producendo progressivamente l'albero di analisi delle dichiarazioni del programma.

Tuttavia, la correttezza grammaticale non è sufficiente a garantire che un programma sia significativo o eseguibile. Esistono proprietà dei costrutti del programma che vanno oltre le capacità espressive dei modelli *context-free* utilizzati per il parsing. È qui che interviene l'analisi semantica.

La fase di analisi semantica ha il compito di arricchire l'AST con informazioni sul significato e di effettuare controlli per garantire che il programma sia semanticamente coerente secondo le regole del linguaggio sorgente. Mentre la fase di analisi sintattica si occupa della forma del programma, l'analisi semantica si occupa del suo significato.

La fase di analisi semantica dell'input dipende dalla semantica del linguaggio sorgente, mentre la fase di sintesi dell'output (traduzione o generazione di codice) dipende sia dalla semantica del linguaggio sorgente sia dalla semantica del linguaggio target. A causa della complessità intrinseca nella definizione formale della semantica di un linguaggio, queste fasi non possono essere automatizzate con la stessa facilità dello *scanning* e del *parsing*.

Per produrre l'output il nostro *transpiler* adotta l'approccio classico: l'AST viene costruito durante il parsing e poi utilizzato per i controlli semantici e la successiva traduzione.

5.1 Inferenza dei Tipi nel Transpiler

L'inferenza dei tipi è un processo cruciale nell'analisi semantica, specialmente per un linguaggio dinamicamente tipizzato come LUA che si intende tradurre in un linguaggio staticamente tipizzato come il C.

Lo scopo è dedurre, per quanto possibile e in fase di "compilazione" (transpilazione), il tipo di dato che un'espressione o una variabile assumerà durante l'esecuzione. Questo è fondamentale per generare codice target corretto e per effettuare controlli di coerenza.

Nel transpiler, l'inferenza dei tipi è primariamente implementata nella funzione `eval_expr_type(struct AstNode *expr, struct symlist *current_scope)`, che restituisce una struttura `struct complex_type`.

La funzione opera ricorsivamente sull'Abstract Syntax Tree (AST) dell'espressione e utilizza la tabella dei simboli corrente per dedurre i tipi.

Nel caso di **valori letterali** (`VAL_T`), il tipo è immediatamente determinabile:

```
result.type = expr->node.val->val_type;
result.kind = CONSTANT;
```

Ad esempio, 123 sarà `INT_T`, "hello" sarà `STRING_T`, `true` sarà convertito in `BOOLEAN_T` (tramite `eval_bool`), e `nil` in `NIL_T`.

Per i **costruttori di tabelle** (`TABLE_NODE_T`), il tipo sarà sempre:


```
result.type = TABLE_T;
result.kind = DYNAMIC;
```

Il contenuto della tabella, infatti, può variare dinamicamente.

Nel caso delle **variabili** (VAR_T), si effettua una lookup nella tabella dei simboli:

```
symbol = find_syntab(current_scope, expr->node.var->name);
```

Se il nome è esattamente "io.read", viene inferito direttamente come FUNCTION_T, per via della gestione speciale nel parser.

Se il simbolo esiste:

```
result.type = symbol->type;
```

Altrimenti si assume:

```
result.type = NIL_T;
```

result.kind resta comunque DYNAMIC.

Per le **chiamate di funzione** (FCALL_T):

Se la funzione chiamata è 'io.read', il tipo di ritorno viene inferito in base agli argomenti forniti. In assenza di argomenti, viene restituito 'STRING_T', corrispondente al comportamento predefinito '"*l"'. Se l'argomento è '"*n"', il tipo restituito sarà 'NUMBER_T'. Nel caso in cui l'argomento sia '"*l"', '"*a"' o '"*L"', il tipo sarà comunque 'STRING_T'. Se viene passato un argomento numerico (ad esempio un intero N), si assume ancora 'STRING_T' come tipo di ritorno. In tutti gli altri casi, il tipo inferito è 'ERROR_T', a indicare un errore nell'inferenza o un uso non gestito.

Se la funzione è **print**, il tipo restituito è sempre:

```
result.type = NIL_T;
```

Per tutte le altre funzioni, l'inferenza del tipo di ritorno segue un ordine di priorità. Si tenta innanzitutto di utilizzare il tipo di ritorno precalcolato presente nel nodo AST, accessibile tramite 'expr->node.fcall->return_type'. Se questo valore non è valido o è pari a 'NIL_T', si effettua una ricerca nella tabella dei simboli corrente tramite la funzione 'find_syntab', cercando di estrarre il tipo registrato per quella funzione. Se anche questa ricerca fallisce oppure non fornisce informazioni sufficienti, si emette un warning e si assume come tipo di ritorno il valore 'USERDATA_T', usato come tipo generico o opaco per rappresentare un'espressione il cui tipo non può essere determinato staticamente. Il kind resta DYNAMIC.

Per **espressioni complesse** (EXPR_T), l'inferenza si basa sull'operatore:

Operatori aritmetici (+, -, *, /) restituiscono:

```
result.type = NUMBER_T;
```

Operatori logici/confronti (and, or, not, ==, =, <, <=, >, >=) restituiscono:

```
result.type = BOOLEAN_T;
```

L'unario -expr ritorna lo stesso tipo di expr, tramite:

```
return eval_expr_type(expr->node.expr->r, current_scope);
```

Le parentesi (`expr`) non influenzano il tipo.

Anche per le assegnazioni (`var = expr`), si ritorna il tipo della parte destra.

Infine, per nodi non riconosciuti si assume:

```
result.type = USERDATA_T;
```

E se `expr` è `NULL`, si ritorna:

```
result.type = NIL_T; result.kind = CONSTANT;
```

Questo sistema consente al transpiler di eseguire controlli di tipo coerenti con un paradigma statico, pur mantenendo la flessibilità necessaria per interpretare correttamente il comportamento dinamico del linguaggio LUA.

5.2 Gestione Semantica di `io.read`

La funzione `io.read` in LUA ha una semantica particolare che dipende dagli argomenti con cui viene chiamata. Il transpiler implementa controlli semantici specifici per `io.read` all'interno della funzione:

```
check_fcall(struct AstNode *func_expr, struct AstNode *args)
```

Questa funzione viene invocata dal parser ogni volta che viene riconosciuta una chiamata di funzione (`func_call`).

Gli obiettivi principali sono:

- Validare gli argomenti passati a `io.read` per assicurarne la conformità alla semantica di Lua.
- Emettere avvisi o errori utili alla diagnosi semantica o alla traduzione.

Il primo blocco `if` è dedicato a riconoscere la chiamata a `io.read`:

```
if (func_expr->nodetype == VAR_T &&
    strcmp(func_expr->node.var->name, "io.read") == 0)
{
    // E' una chiamata a io.read
    // ... (logica di controllo) ...
}
```

La condizione verifica che l'espressione sia una variabile con nome esattamente `"io.read"`. Questo è possibile grazie alla normalizzazione eseguita durante l'analisi sintattica tramite la funzione `new_io_read_identifier_node`.

Se `args` è `NULL`, la chiamata è `io.read()` — un caso valido, che equivale a `io.read("*l")` (lettura di una linea).

Se presente, l'argomento viene validato:

```
{
    yyerror("io.read: argument must be a literal format string or number");
}
```

Il transpiler impone che l'argomento sia un letterale, per semplificare la traduzione.

```

if (current_arg->node.val->val_type == STRING_T)
{
    const char *fmt = current_arg->node.val->string_val;
    if (strcmp(fmt, "%n") == 0 || strcmp(fmt, "%l") == 0 ||
        strcmp(fmt, "%a") == 0 || strcmp(fmt, "%L") == 0)
    {
        // Formato valido
    }
    else
    {
        yyerror("io.read: argument must be a format string or a number");
    }
}

```

Sono accettati solo i formati standard: "%n" (leggi un numero), "%l" (leggi la prossima linea), "%a" (leggi il resto del file), "%L" (leggi la prossima linea mantenendo il terminatore di linea).

```

else if (current_arg->node.val->val_type == INT_T ||
        current_arg->node.val->val_type == FLOAT_T)
{
    if (current_arg->node.val->val_type == FLOAT_T)
    {
        yywarning("io.read: numeric format is float, will be treated as integer");
    }
}

```

Valori interi e floating point sono interpretati come numero di byte da leggere. I float generano un warning: saranno trattati come interi.

```

else
{
    yyerror("io.read: argument must be a format string or a number");
}

```

Tipi diversi da stringa o numerico (come booleani o nil) non sono accettati.

```

if (current_arg->next != NULL)
{
    yywarning("io.read: multiple arguments provided. Translation to C might only support the first or be complex.");
}

```

Se vengono forniti più argomenti, viene emesso un avviso. Lua considera solo il primo, ma il transpiler potrebbe non supportare correttamente argomenti multipli, o gestirli in modo semplificato.

5.3 Inferenza del Tipo di Ritorno delle Funzioni

```
infer_func_return_type(struct AstNode code, struct symlist func_scope)
```

Questa funzione viene chiamata tipicamente al termine dell'analisi sintattica di una definizione di funzione (nella regola `func_definition` di `parser.y`). Prende in input il nodo AST che rappresenta il corpo della funzione (`code`, che è una lista concatenata di statement) e la tabella dei simboli dello scope interno alla funzione (`func_scope`).

L'obiettivo è analizzare tutte le possibili istruzioni `return` all'interno del corpo della funzione per dedurre un tipo di ritorno il più preciso possibile.

La funzione opera iterando attraverso la lista di statement (`current = code; while (current) ...`) che compongono il corpo della funzione e analizzando ricorsivamente i blocchi di codice annidati.

Inizializzazione

- `inferred_type = NIL_T`; — Tipo di ritorno inizializzato a `NIL_T`, usato come default.
- `return_count = 0`; — Contatore delle istruzioni `return` incontrate.
- `first_return_type = NIL_T`; — Memorizza il tipo del primo `return` significativo.

Analisi delle Istruzioni RETURN_T Se lo statement corrente è di tipo `RETURN_T`:

- Si incrementa `return_count`.
- Si valuta il tipo dell'espressione restituita con `eval_expr_type(current->node.ret->expr, func_scope).type`. Se non presente, rimane `NIL_T`.
- Se è il primo `return`, il tipo diventa sia `first_return_type` che `inferred_type`.
- Nei `return` successivi:
 - Se il tipo corrente e quello del primo `return` sono entrambi numerici (`INT_T`, `FLOAT_T`, `NUMBER_T`), si generalizza a `NUMBER_T`.
 - Se `first_return_type` era `NIL_T` e quello corrente no, si aggiorna `inferred_type`.
 - Se il tipo corrente è `NIL_T` e il primo no, non si modifica `inferred_type`.
 - Se i tipi sono incompatibili e non numerici, si emette un warning e si restituisce subito `USERDATA_T`.

Analisi Ricorsiva di Blocchi Strutturati Poiché le istruzioni `return` possono trovarsi all'interno di blocchi condizionali o cicli, la funzione `infer_func_return_type` viene chiamata ricorsivamente per analizzare il corpo di tali blocchi.

Blocchi IF_T:

- Si inferisce il tipo di ritorno per il blocco `then` (`if_type`) e per l'eventuale blocco `else` (`else_type`).
- Se `inferred_type == NIL_T`, e uno dei due è non `NIL_T`, lo si usa come tipo inferito.
- Se entrambi sono numerici, si imposta `NUMBER_T`.
- Se sono diversi e non compatibili, si emette un warning e si imposta `USERDATA_T` (solo se `inferred_type` è ancora `NIL_T`).

Blocchi FOR_T:

- Si inferisce il tipo del corpo del ciclo (`for_type`).
- Se `for_type` è diverso da `NIL_T` e `inferred_type` è ancora `NIL_T`, lo si aggiorna.

6 Traduzione

L'ultima fase del transpilatore è la traduzione dalla struttura intermedia (AST) al linguaggio obiettivo, ossia C. Dopo la traduzione sono generati due file: un `.c` contenente la traduzione vera e propria del condice in input scritto in Lua, e un `.h` in cui sono contenute definizioni di funzioni utili alla traduzione, in particolare:

```
char* c_lua_io_read_line(){
    char *buff;
    scanf("%ms", &buff);
    return buff;
}

float c_lua_io_read_number(){
    float ret;
    scanf("%f", &ret);
    return ret;
}

char *c_lua_io_read_bytes(int n)
{
    char *buff = malloc(sizeof(char) * (n + 1));
    scanf("%ms", &buff);
    buff[n] = '\0';
    return buff;
}

typedef struct
{
    char *key;
    union value
    {
        int int_value;
        double float_value;
        char *string_value;
        bool bool_value;
    } value;
} lua_field;
```

Le funzioni `c_lua_io_read_line`, `c_lua_io_read_number` e `c_lua_io_read_bytes` sono utili all'implementazione dell'input dallo `stdin` in base ai diversi tipi di dato. Lo struct `lua_field` è definito per permettere la traduzione della table in Lua come una hashmap in C. Infatti questa struct rappresenta il singolo campo della tabella da tradurre mentre l'oggetto stesso della tabella è tradotto come un array di `lua_field`.

Inoltre nel file `.h` sono dichiarate le funzioni definite nel file tradotto `.c` e sono importate le librerie utili alla gestione di I/O, funzioni di malloc e Bool.

La traduzione dei file sorgenti segue il seguente flusso:

- la funzione `translate`, preso in nodo radice itera tutti i nodi figli ricorsivamente tenendo in considerazione il contesto;
- la funzione `translate_node` traduce il nodo in questione in base alle informazioni contenute nell'AST e nella Symbol Table;

- la funzione `translate_params` traduce i parametri di una funzione esplicitandone i tipi; a differenza della funzione `translate_list` che traduce sì i parametri ma senza dichiararne i tipi, usata nella chiamata a funzione.

6.1 Test

I file di test sono organizzati nella cartella **test**. I test sono organizzati gerarchicamente mediante una suddivisione in sottocartelle che individuano l'oggetto del test. A sua volta, in ogni sottocartella, distinguiamo i test validi, dagli errori. Mediante il comando :

```
>> make test
```

è possibile testare tutti i file all'interno della cartella `test` e in particolare quelli considerati validi. Con il comando:

```
>> make error
```

vengono invece testati tutti i file contenenti errori lessicali e/o sintattici così da testare la robustezza del transpilatore; vengono stampati in console tutti gli errori rilevati.

Bibliografia

- [1] Bison reference manual.
- [2] C reference manual.
- [3] Flex reference manual.
- [4] Lua reference manual.
- [5] Floriano Scioscia. Formal languages and compilers. 2025.