



UNIVERSITÀ
DEGLI STUDI
DI BRESCIA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea

in Ingegneria Informatica

Relazione Finale

PROGETTO E SVILUPPO DI UN INTERPRETE DEL LINGUAGGIO DI PROGRAMMAZIONE SIMPLA

Relatore: Chiar.mo Prof. Gian Franco Lamperti

Laureando:

Sartorelli Marco

Matricola n. 719788

Anno Accademico 2020/2021

Sommario

Introduzione	5
Obbiettivi e attività della tesi	7
Cenni di teoria	8
Differenza interprete compilatore [1].....	8
Analisi lessicale	9
Analisi sintattica.....	10
Analisi semantica.....	12
Generazione di codice intermedio	14
Interpretazione.....	15
Simpla	16
Caratteristiche di alto livello.....	16
Tipi di dati	16
Struttura di un possibile programma	17
Body	18
Espressioni.....	20
Operatori	20
Fattori	21
Puntatori	22
Analisi lessicale.....	24
Cenni di espressioni regolari	24
Flex	25
Dichiarazioni	25
Regole di traduzione	26
Funzioni ausiliarie	27
Codice C generato	27

Variabili e funzioni [5]	27
Implementazione.....	28
Strutture	28
Cattura e gestione errori	29
Analizzatore sintattico	30
YACC	30
Dichiarazioni	30
Regole di traduzione	30
Funzioni ausiliarie	31
Implementazione.....	31
Albero sintattico astratto.....	31
Strutture	32
Cattura e gestione degli errori	38
Analizzatore semantico.....	39
Cenni di hashtable.....	39
Implementazione.....	41
Strutture	41
Type-checking e controllo semantico	43
Cattura e gestione degli errori	46
Fase di esecuzione : Interpretazione	47
Cenni di stack-based runtime enviroment	48
Implementazione.....	50
Strutture	50
Esecuzione	52
Conclusione	64
Appendice	65
Grammatica BNF	65

Esempio codice : Fibonacci	67
Esempio codice : Puntatori.....	67
Riferimenti.....	68

Introduzione

Un computer è una macchina che esegue istruzioni appartenenti all'ISA della sua architettura. Questo procedimento era valido in passato, quando il computer nacque, ed è valido ancora oggi a distanza di molti anni.

Il problema dell'utilizzo di istruzioni di così basso livello è che sono state create avendo in mente e dovendo far conto alle unità fisiche che formano le componenti di calcolo della macchina e non per rappresentare quello che per esempio può essere il modo di descrivere le espressioni in matematica.

Con la maggior disponibilità di potenza di calcolo e l'aumentare della complessità dei sistemi in gioco sono venuti a crearsi dei linguaggi di programmazione che avevano lo scopo di astrarre quello che era il machine code e permettere a chi sviluppava di poter scrivere codice in maniera più leggibile per una persona. Il funzionamento a basso livello non cambia, si basa sempre sull'esecuzione di istruzioni che l'unità di calcolo ci fornisce, però tramite quelli che sono compilatori e interpreti possiamo trasformare un linguaggio in un altro o eseguire direttamente un certo linguaggio permettendo l'astrazione sopra indicata.

L'interprete di un linguaggio di programmazione è un programma che esegue codice sorgente di un linguaggio attenendosi alle specifiche dello stesso. Ovvero è un applicativo che preso in input il codice sorgente lo esegue sulla macchina su cui lui è in running.

Apriamo una parentesi sulla compatibilità di un linguaggio con una certa architettura.

In generale architetture diverse hanno codice macchina diverso e quindi l'attività di interpretazione potrebbe dover essere differente, però il tutto dipende dal linguaggio con il quale l'interprete viene scritto.

Ad esempio nel nostro caso è scritto in C quindi Simpla sarà compilabile su qualsiasi macchina su cui il C sia implementato (Linux, Windows, MacOS..).

Andando ad esplorare il funzionamento del progetto molto di ciò che si vedrà riguarda l'implementazione del linguaggio che è distinta dal linguaggio in sé. Cose come stack, discesa ricorsiva, sono caratteristiche particolari che

un'implementazione potrebbe usare, non sono importanti particolari richiesti per il rispetto della specifica del linguaggio.

Obbiettivi e attività della tesi

Per quando lo scopo di un interprete e il suo funzionamento “black-box” sia chiaro ai molti ne è spesso sconosciuto l’interno o come le sue varie parti collaborino per la riuscita dell’operazione.

Scopo della tesi è mostrare una possibile implementazione spiegandone gli strumenti utilizzati, le possibili alternative, quali sono i meccanismi interni che lavorano tra loro e perché.

Come primo passo verrà esplorato il linguaggio Simpla, i suoi costrutti, la grammatica e alcuni esempi di possibile utilizzo.

Da lì in poi parleremo di come sia stato possibile, sfruttando gli strumenti esterni Lex e Yacc, implementare in codice C l’interprete. Infine, per chiarire il funzionamento verranno esplorati i flussi di esecuzione che si hanno nell’interpretazione di alcuni possibili esempi di codice Simpla.

Divideremo la spiegazione in capitoli che seguono quello che è stato anche lo sviluppo del progetto e che si attengono alla teoria spiegata in aula.

Inoltre verrà specificato quali parti sarebbero potute essere completamente staccate tra di loro (ovvero quali avevano un accoppiamento vicino allo 0) e quali invece lavorano a strettissimo contatto tanto da poterle vedere quasi come un unico componente.

Un importante osservazione da fare riguarda la somiglianza tra i costrutti e le modalità con cui opera Simpla alle controparti di C.

Seppure ovviamente in maniera molto più semplicistica si può considerare Simpla come un linguaggio C-like e grazie a questo esplorarne i funzionamenti interni può portare a capire leggermente meglio come ragiona il compilatore C e tutti quelli simili per quanto riguarda le operazioni base.

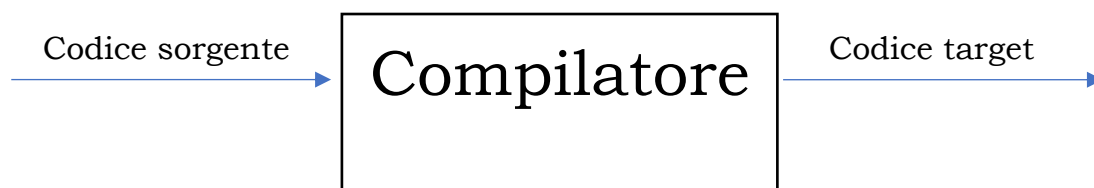
Cenni di teoria

Differenza interprete compilatore [1]

È stata data nell'introduzione una piccola definizione di cosa sia un interprete, riprendiamo qui il concetto per spiegarlo meglio ed esporre le differenze che presenta con un compilatore.

Un compilatore è un programma che legge un programma in un linguaggio , il codice sorgente, e lo traduce in un programma equivalente , il codice target.

Un importante ruolo del compilatore è di riferire qualsiasi errore trovato durante il suo processo d'esecuzione.



Se il codice target è machine code eseguibile allora può essere chiamato direttamente e dati eventuali input produce output.

Un interprete invece di produrre codice target come traduzione esegue direttamente le operazioni specificate nel codice sorgente con gli eventuali input dati dall'utente.



Il codice target machine language prodotto dal compilatore è solitamente molto più veloce di un interprete, il quale però può normalmente dare informazioni di

diagnostica errore migliori perché esegue il programma sorgente riga per riga direttamente.

Analisi lessicale

Indipendentemente dalla scelta del programma da implementare il primo passaggio che viene eseguito è l'analisi lessicale.

Il suo funzionamento è la lettura, carattere per carattere, del codice sorgente e il suo compito è duplice : l'associazione di un simbolo ad un carattere o un insieme di caratteri e la rimozione di tutti gli elementi che non aggiungono significato al codice, ad esempio spazi, commenti.

Un simbolo è un'astrazione che racchiude tutte le implementazioni di quel tipo. Ad esempio il simbolo 'intconst' può essere usato per racchiudere tutti i possibili numeri interi che un analizzatore incontra nella sua esecuzione.

Se il lavoro passato allo stadio successivo consistesse solo nel simbolo associato alla stringa letta però andrebbero perse tutte le 'istanze' del simbolo stesso, per questo motivo viene anche definito un attributo lessicale.

Un attributo lessicale (che può essere anche vuoto in certi casi) contiene l'istanza del simbolo che viene riconosciuto.

Riprendendo l'esempio fatto sopra quando viene riconosciuto un simbolo 'intconst' assoceremo ad esso il numero specifico che è stato letto.

Un concetto importante che deve essere rispettato perché tutto funzioni come dovrebbe è il maximal munch.

Definito come il prefisso più lungo dell'input rimanente che è un simbolo del linguaggio [2] corrisponde al massimo numero di caratteri leggibili dall'input affinché la stringa letta fino a quel punto sia ancora almeno un simbolo del linguaggio.

Se non venisse rispettato il numero di linguaggi analizzabili sarebbe drasticamente ridotto perché se esistessero due simboli dei quali il primo è la definizione del secondo più qualcosa allora questo primo simbolo non sarebbe mai riconoscibile in quanto verrebbe sempre selezionato il secondo essendo quello che soddisfa con meno caratteri una regola.

Esempio del funzionamento con una frase del linguaggio Simpla

N	U	M		:		I	T	E	G	E	R	;
---	---	---	--	---	--	---	---	---	---	---	---	---

Anche se non ancora spiegato il significato è abbastanza intuitivo, stiamo dichiarando una variabile di nome “num” intera.

Il funzionamento come spiegato prima prevede la lettura graduale dei caratteri in input finché almeno una delle regole lessicali è ancora valida.

Immaginiamo una regola che identifica le variabili e accetta tutti i caratteri alfanumerici , allora la nostra stringa finale (maximal munch) sarà ‘NUM’.

N	U	M		:		I	T	E	G	E	R	;
---	---	---	--	---	--	---	---	---	---	---	---	---

Chiamando ID il simbolo che descrive le variabili, il token finale che si forma è <ID,”NUM”> dove il secondo campo è l’attributo lessicale.

Continuando con la lettura si incontra lo spazio che come spiegato a inizio sezione non è un carattere utile alle successive operazioni e viene semplicemente ignorato

Scorrendo viene trovato ‘.’ che avrà una sua regola siccome significativo per la sintassi.

Continuando con lo stesso ragionamento il prodotto finale sarà come segue, integer e ; saranno matchati dalle rispettive regole e la riga finisce.

In realtà la fine della riga è contrassegnata da un carattere speciale ‘\n’ ma è un dettaglio riguardante gli specifici sistemi operativi e lo vedremo successivamente.

N	U	M		:		I	T	E	G	E	R	;
---	---	---	--	---	--	---	---	---	---	---	---	---

Analisi sintattica

Questa sezione usufruisce dei token passati da quella precedente e tramite le produzioni della grammatica libera dal contesto decide se la frase è legale per il linguaggio.

Una grammatica non contestuale è un insieme di regole che definiscono tutte le possibili frasi sintatticamente corrette per quel linguaggio.

Es. $A \Rightarrow -A$, e si dice che A deriva -A.

Una derivazione è proprio l'operazione con la quale si sostituisce al simbolo RHS quello o uno di quelli LHS.

Una derivazione in uno o più passaggi dall'assioma della grammatica forma una frase del linguaggio.

Es. $A \Rightarrow A \mid -A \mid (A) \mid id$

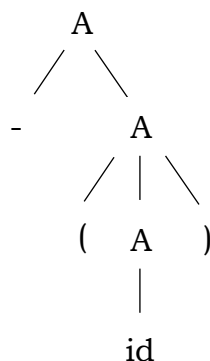
$A \Rightarrow -A \Rightarrow -(A) \Rightarrow -(id)$, allora possiamo dire che $-(id)$ è una frase del linguaggio.

Un metodo alternativo per la rappresentazione di stringa accettata è tramite albero sintattico.

Un albero sintattico è una rappresentazione grafica di una derivazione indipendentemente dall'ordine scelto nelle sostituzioni [3]

Se la grammatica non è ambigua l'albero è unico per ogni derivazione, nel caso non lo fosse le soluzioni sono due: eliminare l'ambiguità operando sulla grammatica non contestuale, scegliere caso per caso quale degli alberi possibili si vuole. La prima è chiaramente più pulita da un punto di vista teorico.

Vediamo come si presenterebbe l'esempio di prima con questo nuovo metodo.

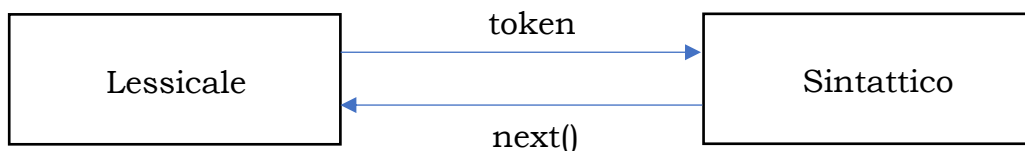


Si vede che le foglie contengono ancora la frase ottenuta tramite derivazione nel passaggio precedente.

Analizzatore lessicale e sintattico sono due sezioni molto accoppiate tra di loro, tanto che si potrebbero unire.

La divisione però rende più efficiente l'operazione e toglie lavoro inutile all'analizzatore sintattico che si dovrebbe sennò preoccupare anche dei caratteri che prima abbiamo detto venir scartati come lo spazio e i commenti.

Schematizziamo il funzionamento :



L'analizzatore sintattico chiede all'occorrenza il simbolo successivo, sarà l'analizzatore lessicale che seguendo le direttive interne incontrerà prima o poi un'azione di return e restituirà il token.

Lo scambio di messaggi è quindi continuo e le esecuzioni non completamente sequenziali.

Analisi semantica

Arrivati a questa fase la conoscenza della struttura sintattica del codice è già nota, il compito è calcolare tutte le informazioni che sono necessarie per la compilazione ma non sono ricavabili dall'analisi sintattica.

Un esempio particolare è se la chiamata di una certa funzione è preceduta dalla sua dichiarazione, in teoria è un vincolo strutturale e dovrebbe essere controllato precedentemente ma l'utilizzo di una grammatica non contestuale per la verifica della correttezza sintattica non permette il controllo di errori appunto basati sul contesto.

Siccome questo passaggio è eseguito prima della compilazione la sua efficacia dipende da quanto può sapere e questo aumenta con l'aumentare della staticità del linguaggio.

Ad esempio un linguaggio fortemente tipizzato come il java può catturare più

errori in fase semantica di un linguaggio come javascript dove il tipo è deciso a tempo di esecuzione.

Una delle azioni più frequenti in questa fase è la costruzione di una symbol table per tenere traccia di nomi e tipi delle strutture in gioco (variabili, funzioni e altro se presente).

Una symbol table è una semplice tabella avente diversi campi possibili a seconda del linguaggio ma in generale quelli sempre presenti sono nome della struttura che stiamo seguendo, il suo tipo e la specifica di cosa sia (variabile, parametro, funzione...).

Inoltre è comune, anche basandosi sulla struttura di prima, controllare la compatibilità dei tipi di espressioni e istruzioni presenti nel codice (type checking).

Generazione di codice intermedio

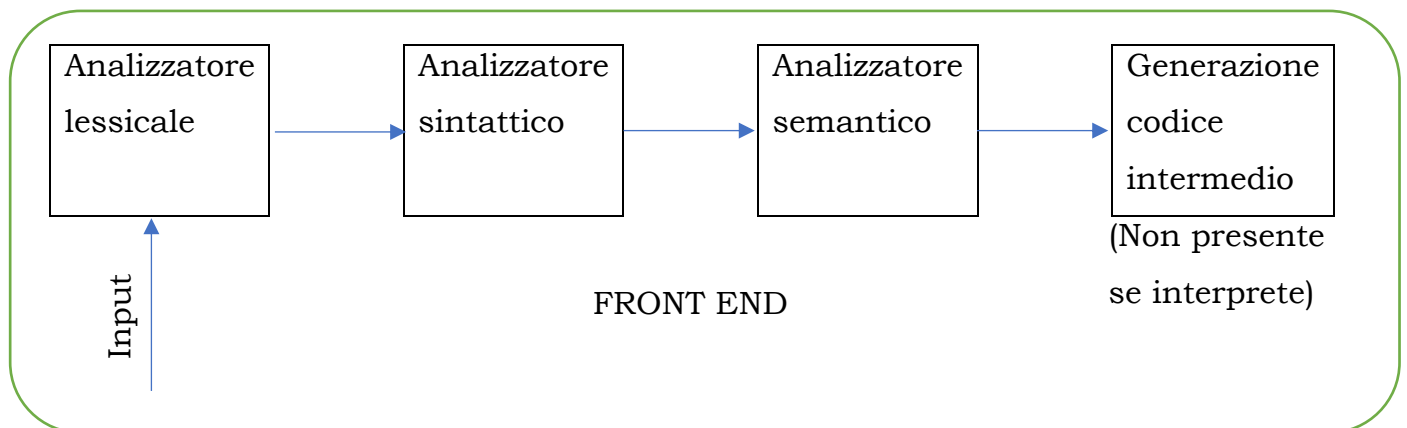
È da questo punto in poi che la struttura del programma cambia a seconda che si voglia creare un compilatore o un interprete.

Per come è stato definito prima un compilatore ha come risultato finale la creazione di un codice, il codice target che può essere di qualsiasi tipo anche se normalmente è di basso livello o addirittura linguaggio macchina.

Mentre un interprete deve eseguire gli statement del codice di partenza uno alla volta direttamente senza produrre necessariamente prodotti intermedi.

Nel progetto, che è un interprete, non è presente questa sezione che però riporto in maniera veloce per completezza.

La generazione di codice è l'ultimo stadio del front-end di un compilatore che è composto in questo modo :



La divisione in front-end, che gestisce tutti i passaggi fino alla generazione di un codice intermedio indipendente dalla macchina target, e back-end, che ottimizza il codice intermedio e genera il codice target, è dovuta alla presenza di diverse architetture target e la volontà di non dover fare un compilatore completo per ognuna di esse.

Dividendo tutte le parti indipendenti dalla macchina target da quelle dipendenti si può creare un back-end dedicato per ogni architettura riutilizzando lo stesso identico front-end tutte.

Il codice intermedio può essere quindi visto come un codice target per un'architettura immaginaria, ideale.

Sarà poi successivo il compito di creare una macchina virtuale che accetti questo

codice e lo traduca in operazioni per la cpu reale.

Interpretazione

Se il programma è un interprete la parte di generazione di codice è completamente saltata, ci troviamo con codice corretto semanticamente e che possiamo visualizzare ad esempio con un albero sintattico.

Quello che rimane da fare è eseguire gli statement uno a uno, per farlo le tecniche sono molteplici, nel progetto è stata usata una stack-based che sarà esplorata più avanti.

Simpla

Caratteristiche di alto livello

- Paradigma imperativo
- Funzioni non innestabili ma ricorsione permessa
- Passaggio di parametri permesso sia per valore che per indirizzo
- Commenti che iniziano con '#' fino al newline
- Le variabili del main sono globali
- Ogni funzione può chiamarne una qualsiasi altra, non c'è nessun ordine di dichiarazione da rispettare
- Funzioni interne a disposizione : read(param-list), write(expr-list),writeln(expr-list)

Tipi di dati

Abbiamo cinque tipi di dati possibili, di cui solo quattro applicabili alle variabili. Integer, real, string, boolean, void (solo per le funzioni).

Ogni variabile è di un solo tipo che è definito quando questa viene dichiarata e non può cambiare.

Es.

a: integer; a = 5;	a: real; a = 15.15;	a: string; a = "stringa";	a: boolean; a = true	
func a():integer result : integer; body #достuff return result; end;	func a():real result : real; body #достuff return result; end;	func a():string result : string; body #достuff return result; end;	func a():boolean result : boolean; body #достuff return result; end;	func a():void body #достuff [return;] end;

Legenda : [] indicano l'omissibilità del corpo.

Una variabile che inizia con “*” è automaticamente considerata un puntatore del tipo che segue (*a : integer), il suo comportamento viene spiegato in un paragrafo apposta.

Struttura di un possibile programma

Nell’appendice ho riportato la grammatica BNF del linguaggio che è il modo migliore per capire quale sia la struttura consentita del codice.

Per comprendere a grandi linee faccio qui qualche esempio di possibile programma ammissibile.

La prima parte (se c’è) è la definizione delle variabili globali.

```
a,b,c.. : type;
```

```
primo,secondo : integer;
```

```
nome : string;
```

```
sposato : boolean;
```

Seconda parte (se c’è) è la definizione delle varie funzioni

```
Func nomeFunzione(param-list):type
```

```
[variabili locali]
```

```
body
```

```
    corpo;
```

```
end;
```

```
Func somma(a:integer, b:integer):integer
```

```
result: integer;
```

```
body
```

```
    result = a+b;
```

```
    return result;
```

```
end;
```

Terza e ultima parte il main

Qui troviamo il corpo principale e il primo ad essere eseguito (la sua presenza è obbligatoria).

La struttura è uguale al corpo di una qualsiasi funzione tranne per il fatto che l'end di chiusura ha un '.' Invece di un ';' .

Prima di esplorare il resto del linguaggio riporto che nell'appendice sono presenti un paio di programmi 'tipo' che usando un po' tutti i costrutti a disposizione e che provano a dare un'idea più chiara di cosa si possa fare.

Body

Il body è la parte contenente tutte operazioni che produrranno poi un risultato, ogni riga può essere uno dei seguenti statement :

- Assign : Operazione di assegnamento, nella LHS avremo una variabile e nella parte destra un'espressione.

`var = expr;`

`a = 3+5; a = b; a = numero();`

- If : Costrutto if a una o due vie, composto come in C dalla sintassi `if expr then body [else body]`.

Possiamo vedere che sia il ramo vero che l'eventualmente esistente ramo false hanno una sezione di tipo body eseguibile, questo concetto di riutilizzo di costrutti comuni porterà il codice dell'interprete a sfruttare in maniera consistente la ricorsione come vedremo.

- While : Sintatticamente composto come `while expr do body end`.

Loop while a condizione iniziale, il corpo è sempre di tipo body consentendo quindi un while dentro un while ecc.

- For : È definito come `for ID = expr to expr do body end`.

La costruzione è leggermente diversa da quella di C o java così come il funzionamento.

Dopo la keyword for deve esserci necessariamente un assegnamento e per quello che abbiamo detto sopra un assegnamento richiede nella parte destra una variabile.

Non è previsto uno spazio dove viene definito quanto aumentare la variabile , questa sarà sempre aumentata di 1.

Infine ci sarà un controllo semantico che non consente di variare il valore

dell'ID all'intero del body, il motivo è l'evitare di scenari in cui la modifica di questa variabile porti ad un loop infinito del ciclo.

- **Return** : Solo nelle funzioni è possibile chiamare questo costrutto per terminare prima della fine del body, la sintassi varia a seconda del tipo di ritorno della funzione.

Se void il return è opzionale e deve essere vuoto (non seguito da nulla)
return ;

In tutte le altre è obbligatoria la definizione di almeno un return in ogni possibile flow di esecuzione, ovvero in qualsiasi caso la funzione venga eseguita il flow deve passare almeno una volta per un return (con gli if può succedere che alcuni pezzi di codice non vengano eseguiti).

Inoltre la sintassi sarà return expr; dove expr sarà dello stesso tipo del tipo di ritorno del metodo.

- **Read** : semplice funzione con sintassi read(id-list), il numero di argomenti è variabile senza limiti.

Quando chiamata verrà interrogato l'utente perché fornisca un valore per ogni argomento presente.

read(a,b,c);

- **Write** : sintassi simile al read ma qui possiamo passare un'espressione come argomento (cosa possano essere le espressioni viene esplorato più avanti).

write(expr-list);

write("Il numero è : ",a);

- **Chiamata di funzione** : quando una funzione non ritorna niente o il valore di ritorno non è di nostro interesse salvarlo può essere chiamata in uno statement dedicato solo a quello (non come LHS di un'assign, in questo caso sarebbe riconosciuta come expr come vedremo dopo).

La sintassi è nomefunzione(expr-list?); dove '?' indica la possibile non presenza di expr-list.

- **Break** : keyword legale solo all'interno di un ciclo while o for che ne impone l'immediata terminazione e uscita.

Il body è una lista più o meno lunga (ma non vuota) di queste istruzioni, come già detto alcune di queste hanno all'interno un body loro quindi abbiamo un

innesto teoricamente illimitato.

Espressioni

Le espressioni sono una composizione di fattori e operatori, l'unico caso particolare è quando non abbiamo operatori ma un solo fattore.

Expr : fattore op fattore op fattore op ...

Expr : fattore

Operatori

Gli operatori possibili sono:

- Operatori logici (and, or, not), di cui i primi due binari e l'ultimo unario.
Sono applicabili solo a tipi booleani.
La valutazione delle espressioni logiche con and e or è fatto in cortocircuito:
Nell'and se il primo è falso ritorno falso sennò il risultato del secondo fattore.
Nell'or se il primo è vero ritorno vero sennò il risultato del secondo fattore.
Comportamento analogo agli operatori && e || in C.
- Operatori relazionali (==, !=, >, >=, <, <=), tutti binari.
Sono applicabili a qualsiasi tipo.
- Operatori aritmetici (+, -, *, /), tutti binari tranne il '-' che è anche unario (negazione).
Sono applicabili solo ad interi e reali.

Tabella delle precedenze

Operatore	Associatività
and, or	Sinistra
==, !=, >, >=, <, <=	No
+, -	Sinistra
*, /	Sinistra
-, not	Destra

Precedenza crescente

Fattori

I fattori possibili sono :

- (espressione) : questa regola sta solo ad indicare che un'espressione tra parentesi è un fattore possibile.
- ID : una variabile di qualsiasi tipo.
- &ID : l'operatore & è utilizzato nei puntatori, il significato e funzionamento è visto più avanti ma coincide quasi completamente con la controparte C, giusto per dare un'anticipazione.
- Costante : una qualsiasi costante di uno dei tipi permessi (intero, reale, stringa, booleano).
- Chiamata di funzione : la definizione di un costrutto per la chiamata di funzioni era già presente nei possibili statement del corpo del programma, il motivo per cui viene trovato anche qua è che può apparire sia come invocazione stand-alone sia all'interno di una più complessa espressione e deve essere riconosciuto in entrambi i casi.

Si vedrà poi che il funzionamento interno per la gestione di questi casi è pressoché identico.

- Espressione condizionale : composto come segue, `if expr then integer1 else integer2` è una struttura di controllo che permette di avere, in esecuzione, al suo posto un intero, quale dei due definiti dipende della condizione booleana `expr`.
- Casting : In Simpla esistono due casting possibili (integer, real) che hanno la forma di una chiamata di funzione e restituiscono il nuovo valore del tipo richiesto.

`a : integer; b : real;`

`b = real(a);`

La funzione di casting `real` accetta come argomento un numero o variabile intera e quella `integer` viceversa.

Puntatori

L'ultima parte da esplorare è il funzionamento dei puntatori all'interno del linguaggio.

Ogni variabile che inizia con '*' è automaticamente considerata un puntatore, ogni '*' in più aggiunge un livello di profondità senza limiti teorici.

****a : integer** è un puntatore a puntatore di variabile.

Si può indicare dove puntare in due modi diversi:

Se la locazione dove puntare è data da un puntatore dello stesso livello di quello che stiamo modificando semplicemente si pongono le due variabili uguali tra loro.

a, *b, **c, **d : integer

c = d;

In questo modo prendiamo il valore di 'd' che sarà un indirizzo e lo assegniamo a c.

Se invece volessimo far puntare c ad un puntatore di livello inferiore si usa l'operatore '&' che indica che vogliamo la locazione in memoria della variabile che segue.

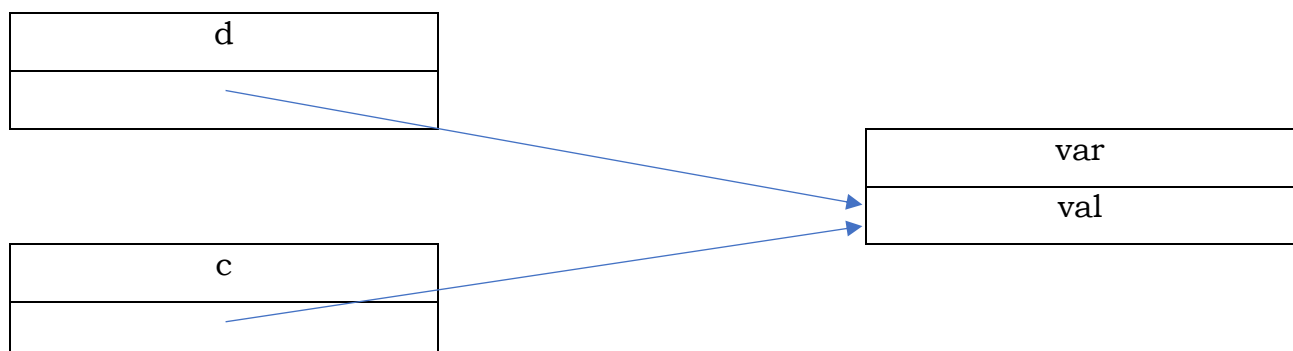
a, *b, **c, **d : integer

b = &a;

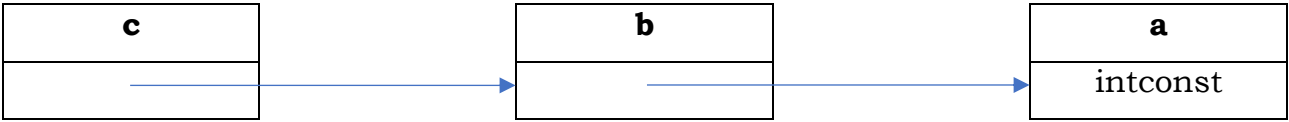
c = &b;

La differenza rispetto a prima è che di 'd' non ottenevamo dove essa stava sulla memoria ma l'indirizzo a cui essa puntava, in pratica 'c' nel primo caso non avrà più bisogno di dopo assegnamento, mentre nel secondo caso c continuerà ad avere bisogno di b per ricavare il valore di a.

Primo caso



Secondo caso



Analisi lessicale

Per la costruzione di questa fase nell'interprete è stato utilizzato lex, un programma che costruisce analizzatori sintattici, in particolare è stata impiegata un'implementazione chiamata flex che è free e open-source.

Cenni di espressioni regolari

Dato l'enorme utilizzo che si ha delle espressioni regolari in questa sezione si riportano qui alcune regole importanti e utilizzate nel progetto.

Un'espressione regolare è un insieme di caratteri che definisce un pattern di ricerca [4].

Qualsiasi carattere matcha se stesso a meno che non abbia un significato particolare per le regex (vedremo dopo alcuni esempi) nel qual caso basta precederlo con '\' per far capire che vogliamo il carattere così com'è non il significato speciale.

Es. a -> matcha tutte le a singole

abba -> matcha tutte le parole che hanno esattamente quelle lettere in quelle posizioni.

Per aggiungere flessibilità al matching vengono introdotti gli operatori quantitativi:

- * indica che possono esserci da 0 a n di ciò che precede.
a* 0 o più a.
- + dev'essercene almeno uno.
Se la regola fosse a*bba allora la stringa "bba" sarebbe accettata, mentre se la regola fosse a+bba no.
- ? 0 oppure 1.

Gli operatori possono anche essere applicati a insiemi di caratteri se vengono circondati da parentesi, (abba)+ accetta la ripetizione una o più volte della parola abba.

Si possono definire insiemi che controllano se una lettera o parola è contenuta nell'insieme. Per farlo si usano le parentesi quadre '[' ']' e all'interno si scrivono gli elementi dell'insieme o il range degli elementi.

[12345] accetta solo questi 5 numeri, potevo anche scriverla come [1-5].

Il range funziona basandosi sui valori ASCII degli estremi.

Gli ultimi due caratteri speciali usati nel progetto sono '.' che può essere visto come un insieme che contiene tutto, in pratica significa che accetta qualsiasi carattere una volta, e '|' che significa or e può essere visto come un insieme con solo due elementi.

Flex

Dato in input un file in formato lex restituisce un analizzatore lessicale in C avente il funzionamento indicato nel file di partenza. Per capire come viene scritto il file descrivo le specifiche di lex e come è stato implementato nel progetto.

La struttura standard prevede tre sezioni :

Dichiarazioni

%%

Regole di traduzione

%%

Funzioni ausiliarie

Dichiarazioni

La prima parte è ulteriormente divisa in due sezioni : black-box, white-box.

Le dichiarazioni black-box sono istruzioni C che verranno copiate così come sono nel file di output e che servono per dare possibilità di usare qualsiasi struttura, dato, funzione di cui si può avere bisogno più avanti nel file (dichiarandole qui o includendole).

Le dichiarazioni white-box sono definizioni regolari, servono per abbinare ad una parola chiave un'espressione regolare che la rappresenta.

Ad esempio per come detto un identificatore di variabile può iniziare o con una lettera e allora tutti i seguenti caratteri (da 0 a n) possono essere alfanumerici, oppure può iniziare con '*' ed essere seguito da un numero illimitato di '*' quando

però questi finiscono deve esserci necessariamente una lettera e da lì in poi 0 fino a n caratteri alfanumerici.

La definizione regolare che soddisfa questa descrizione è :

id `**{letter}({letter} | {digit})*`

Con questa regola associamo ad id il pattern sulla destra.

Il pattern fa uso di elementi che abbiamo visto nel paragrafo precedente ma spieghiamo in dettaglio per chiarezza.

Il backslash indica che il carattere che segue non deve avere il significato speciale che ha nelle espressioni regolari ma dev'essere matchato per come è, il carattere asterisco.

L'operatore quantitativo seguente indica che possono esserci da 0 a n “*”.

Le parentesi graffe non sono parte delle regex ma del mondo delle definizioni regolari e quando inserite hanno tra parentesi il nome di una definizione precedente, ad esempio in questo caso letter e digit saranno precedentemente dichiarate in questo modo :

letter `[A-Za-z]`

digit `[0-9]`

`{letter}` non ha operatori quantitativi ed impone che venga trovata almeno una lettera.

L'or dentro le tonde specifica un insieme a due elementi : lettera e numero.

Infine il carattere speciale “*” viene applicato al contenuto delle tonde per indicare che possono esserci da 0 a n elementi di quell'insieme dopo una lettera.

Le altre definizioni sono abbastanza simili e più semplici di questa e non le riporto, aggiungo solo il caso speciale per catturare i caratteri di newline.

Sistemi operativi diversi hanno diversi modi di indicare una nuova linea :

Windows `\r\n`

Linux `\n`

Per completezza sono stati gestiti entrambi i casi.

Regole di traduzione

Una regola di traduzione è composta da un'espressione regolare ed un'azione ad essa collegata (un frammento in codice C).

Unica alternativa al posto dell'espressione regolare si può avere una definizione regolare (dichiarazioni white-box).

Il frammento in codice C specifica che azioni eseguire quando il pattern associato viene matchato.

Espressione regolare	Azione
<code>{acapo}</code>	<code>{contatore_linea++;}</code>

Con definizione regolare : `acapo` `\n| \r\n`

Funzioni ausiliarie

Parte dedicata all'implementazione di funzioni utili all'esecuzione dell'analizzatore.

Con l'aumentare della complessità si può preferire la scrittura di questa parte in un file esterno, la cosa è possibile e semplicemente fatta seguendo le regole per il collegamento di file di C.

Codice C generato

Come detto l'output di Flex è un analizzatore lessicale scritto in C.

Esternamente questo programma presenta una funzione `yylex()` di tipo intero, quando chiamata inizia a leggere caratteri dallo stream a cui punta la variabile `FILE *yyin` (di default lo standard input) finché riconosce un simbolo a quel punto esegue l'azione associata ad esso.

Essendo una funzione questa continua ad eseguire fino ad un `return` oppure finché non termina la lettura del file in ingresso.

Nel progetto quasi ogni azione termina con un `return`, ciò che viene passato è un `enum` (quindi intero, rispettando il tipo descritto prima), che viene creato nella prossima sezione, esso associa ad ogni simbolo del nostro linguaggio un numero. Le uniche azioni senza `return` e in generale senza nemmeno un'azione sono tutti quei caratteri che devono essere scartati dall'analizzatore lessicale (commenti , spazi) e per come funziona associando ad essi un frammento di codice C vuoto vengono automaticamente semplicemente ignorati e il chiamante della funzione non si accorge di niente.

Variabili e funzioni [5]

Vengono qui riportate variabili e funzioni generate da Flex e usate nel progetto

- **int yylex(void)** : chiamata per invocare il lexer, ritorna un token.
- **char *yytext** : pointer alla stringa matchata.
- **yylen** : lunghezza della stringa matchata.
- **FILE *yyout** : file di output.
- **FILE *yyin** : file di input.

Implementazione

Strutture

Le due cose più importanti che l'analizzatore deve rendere disponibili a fine chiamata sono il simbolo e l'attributo lessicale.

Il simbolo per come è costruito il programma verrà restituito tramite return dal metodo, rimane il problema di come salvare l'attributo.

Value

Per risolvere il problema definiamo una 'union' con possibili valori i tipi permessi nel linguaggio Simpla.

```
typedef union uvalue
{
    int ival;
    float rval;
    char *sval;
    Boolean bval;
}Value;
```

Con Boolean che è un semplice enum contenente {true,false}.

Così facendo quando viene riconosciuto un simbolo che ha bisogno di un attributo lessicale lo si salva in Value con il tipo corretto.

Per rendere fruibile il valore salvato in Value si crea una variabile globale nelle dichiarazioni black-box, ogni volta che viene chiamato yylex() se il simbolo ha un attributo lo si può trovare in quella variabile, verrà poi sovrascritto alla chiamata successiva ma non è un problema.

Cattura e gestione errori

Cosa succede quando un carattere che non soddisfa nessuna regola viene letto?
Se non viene gestito nulla.

Per catturare gli eventuali errori lessicali viene creato un pattern che accetta qualsiasi carattere (‘.’), questa regola viene messa come ultima nel file Flex perché per come funziona internamente se c’è un conflitto tra due regole viene scelta quella scritta per prima (l’errore viene chiamato solo se nessun’altra regola avrebbe potuto gestire quell’input).

```
. {printf("Line %d: ErrLessicale carattere %s non riconosciuto\n",line,yytext);  
    errLessicale();}
```

Nell’azione associata al pattern vengono fatte due cose : comunicazione dell’errore con stringa matchata e linea dove questo succede, chiamata di una funzione che si occupa solo della terminazione prematura del programma C.

Analizzatore sintattico

Anche questa parte del programma è generata tramite uno strumento apposito, in questo caso è Bison, un'implementazione compatibile e leggermente estesa di YACC.

YACC

È uno strumento di generazione di parser, ovvero genera un programma che dati in input simboli e basandosi sulla grammatica non contestuale del linguaggio restituisce un albero sintattico.

La struttura, ugualmente a Lex, prevede tre sezioni

Dichiarazioni

%%

Regole di traduzione

%%

Funzioni ausiliarie

Dichiarazioni

Anche qui come prima si divide ulteriormente in due sezioni: white e black box.

Le dichiarazioni black-box sono frammenti di codice C che verranno copiati tali e quali nel programma generato. Ad esempio inclusione di file esterni, definizione di variabili o costanti...

Le dichiarazioni white-box sono tokens (esiste anche altro, questo è quello importante), ovvero qui vengono definiti i valori enum che come detto l'analizzatore lessicale restituisce in corrispondenza dell'azione di un pattern.

Regole di traduzione

Una regola di traduzione è composta da una produzione della grammatica non contestuale ed un frammento di codice C che è un'azione semantica.

Produzione grammaticale	Azione semantica
type : INTEGER	{ \$\$ = keynode(T_INTEGER); }

Il funzionamento dell'azione semantica viene spiegato più avanti.

Unica alternativa ad un token nella produzione grammaticale è un unico carattere ASCII, in quel caso l'analizzatore lo può passare senza ricorrere ad enum e l'analizzatore sintattico lo indica nella produzione chiudendolo tra apici.

Es. `var_decl : id_list ':' type ';' ;`

Il tutto funziona perché Bison quando genera il programma a partire dal file YACC crea una struttura enum con tutti i tokens dichiarati nel file, ad essi associa numeri crescenti a partire da 258, quindi non c'è collisione con tutti i caratteri ASCII che hanno associato un intero nel range 0-255, e ricordiamo che togliendo simboli e nomi utili alla presentazione sotto sotto è tutto gestito tramite numeri.

Per semplicità si può immaginare che l'enum creato da Bison contenga tutti i caratteri ASCII più i tokens dichiarati nel file YACC.

Funzioni ausiliarie

Parte dedicata a funzioni di supporto per l'azione semantica collegata alla produzione, possono essere, per mantenere il codice più pulito, dichiarate in un file a parte e incluse nelle dichiarazioni black-box.

Implementazione

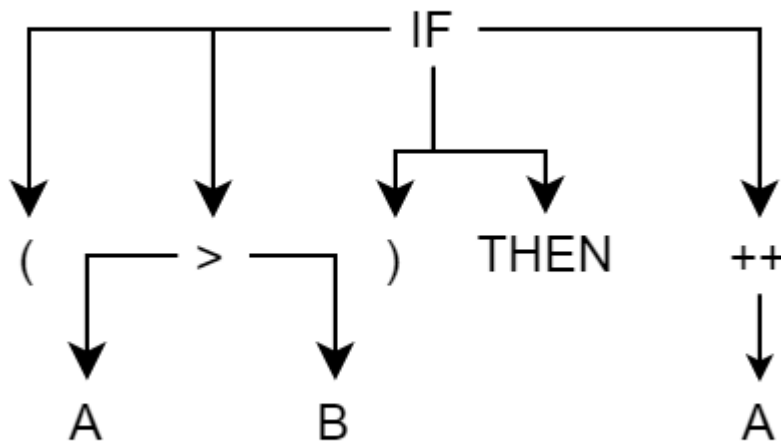
Oltre a dover gestire l'implementazione di questa parte è necessario collegarla a quella precedente, questo succederà anche nelle prossime sezioni ma non con lo stesso livello con cui è presente qui perché come spiegato nei cenni di teoria analisi lessicale e sintattica sono molto vicine tra loro, tanto da poter essere, volendo, un'unica sezione.

Albero sintattico astratto

Dai cenni di teoria sappiamo che un possibile output dell'analizzatore sintattico è l'albero sintattico, però tenendo traccia anche di dettagli di forma spesso risulta uno spreco di memoria.

Es. `if(a>b) then a++;`

Un albero sintattico potrebbe avere questa costruzione



È immediato riconoscere che ‘(’, ‘)’ , ‘then’ , non aggiungono significato all’albero se già sappiamo che è un if, e potremmo eliminarli in favore di un risparmio di memoria.

L’albero che ricaveremmo si chiama albero sintattico astratto, che contiene solo informazioni necessarie per capire cosa quel costrutto vuole fare.

Strutture

Typenode

Enum contenente tutte i terminali del nostro linguaggio.

Un costrutto è terminale se compare solo nella RHS di una produzione della grammatica e mai nella LHS.

Es. T_INTEGER, T_REAL, T_BOOLEAN, T_STRING : sono i quattro valori che servono per identificare di che tipo è una variabile.

T_WRITE, T_WRITELN : identificano le due chiamate per stampare su standard output gli argomenti della funzione.

Uno dei possibili valori è T_NONTERMINAL, viene usato per identificare tutti i costrutti che compaiono almeno una volta nella LHS di una produzione, per capire di quale nonterminale si tratta si usa l’enum Nonterminal.

Nonterminal

Enum contenente i valori dei nonterminali del linguaggio.

Node

L'albero è composto da un insieme di nodi ognuno collegato ad altri tramite una relazione padre-figli.

Ogni nodo è formato dai seguenti campi :

- Typenode : identifica se è un nonterminale o un terminale e in quest'ultimo caso anche che tipo di terminale è.
- Value : struttura già usata nella sezione lessicale, è un'unione che ha come possibili valori i quattro tipi del linguaggio.
Quando il nodo è nonterminale si usa questa variabile per salvare l'enum del tipo di Nonterminal che rappresenta.
- Int linea : contiene il numero della linea del codice sorgente alla quale quel nodo fa riferimento. Ha solo utilizzo per la parte di debugging.
- Node *c1,*c2,*b : tre puntatori a Node che rappresentano i due figli e il fratello.

Albero

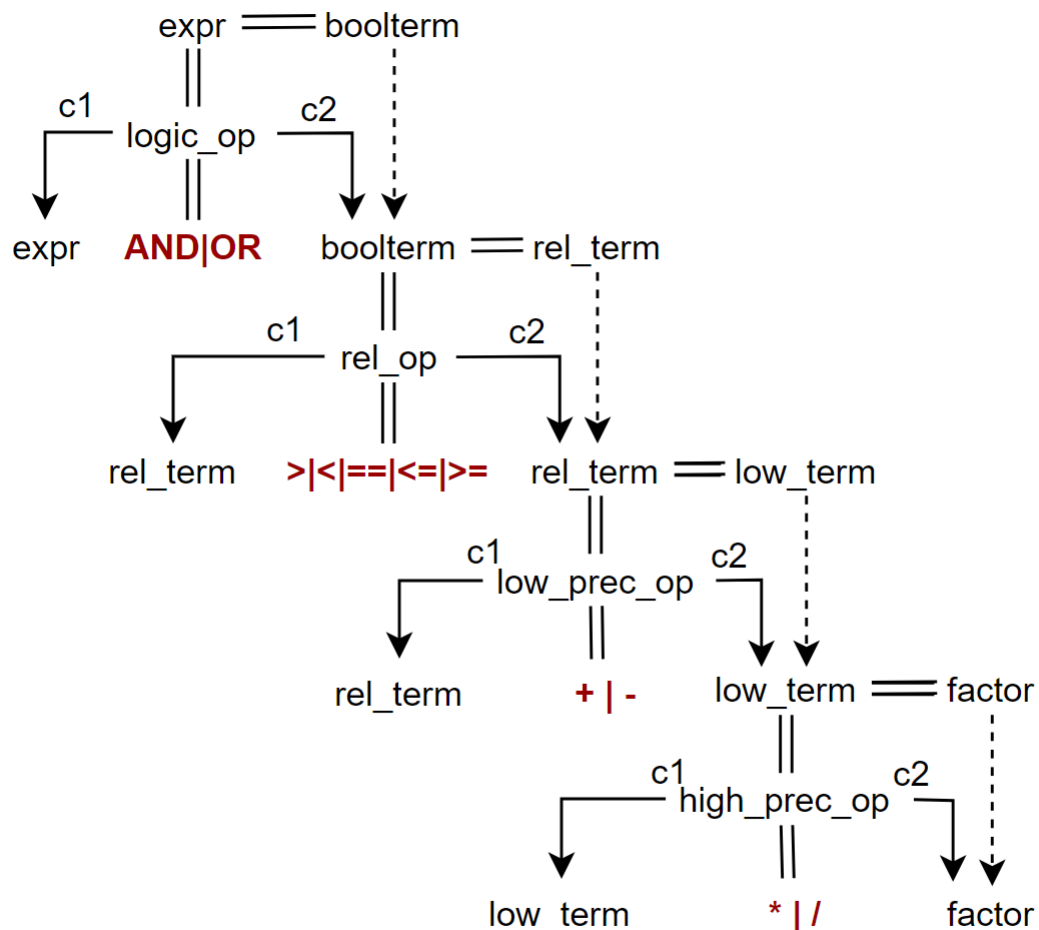
L'albero prodotto dall'analizzatore sintattico del progetto è di tipo astratto.

Come spiegato prima ogni Node ha due figli e un fratello, l'idea dietro questa struttura è che il fratello venga usato per quei costrutti che sono liste, tipo stat-list, dove usiamo questo puntatore per percorrere passo per passo la lista, mentre i due puntatori figli sono per scendere più in profondità in un costrutto nonterminale.

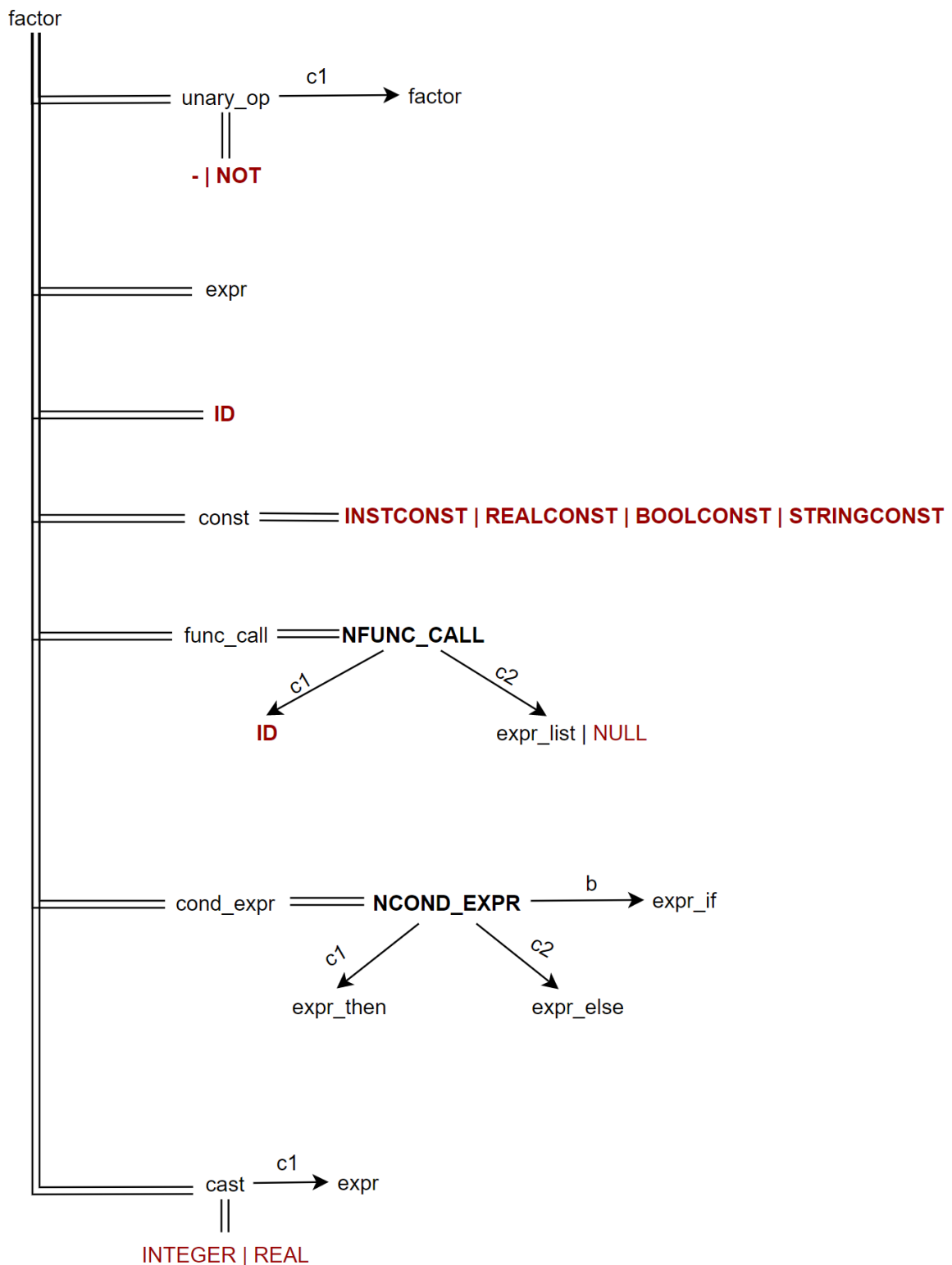
Riporto alcune regole con cui l'albero viene composto nelle azioni semantiche del file YACC che risulta più efficace che spiegarlo a parole, inoltre aggiungo all'appendice un albero completo di un programma d'esempio.

Dato il gran numero di regole presenti la rappresentazione con un solo schema sarebbe poco chiara, le ho quindi spezzate in parti logiche che possono essere unite senza problemi successivamente.

I simboli '=' allungati se singoli indicano che l'albero in quel nodo ha valore uguale alla 'RHS' (al successivo), mentre se sono due indicano i possibili valori che il nodo assume, quale dei vari venga scelto avviene durante l'analisi sintattica in base a cosa arriva dall'analizzatore lessicale.



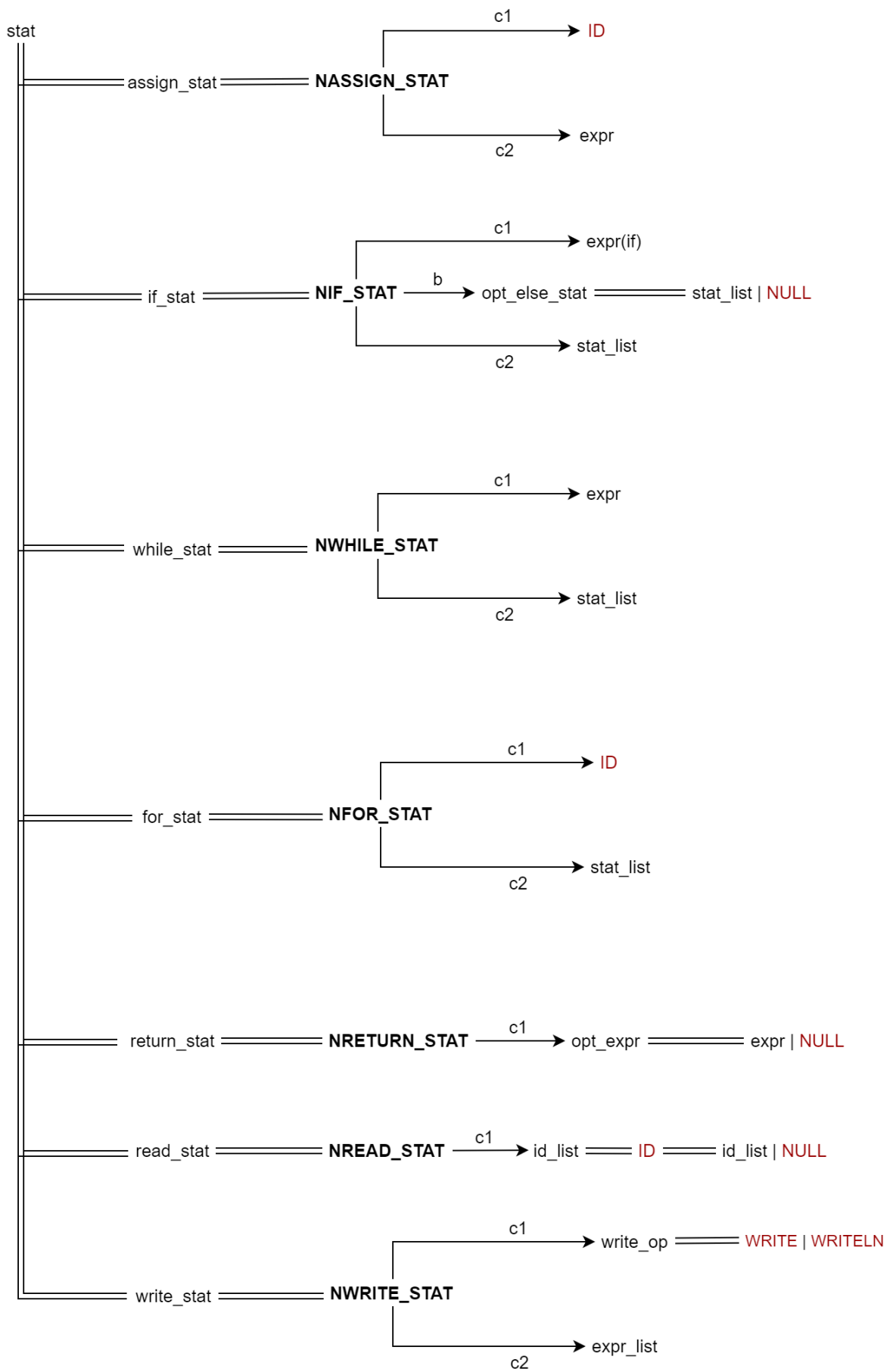
Espando ora la parte factor.



Il comportamento degli '=' allungati è uguale al precedente, si aggiungono le scritte nere maiuscole in grassetto, queste rappresentano i nodi non terminali e vengono usati quando due costrutti diversi possono trovarsi nello stesso contesto

e hanno una costruzione uguale che non permette di riconoscerne uno dall'altro, allora gli diamo un nome che quando visiteremo l'albero negli stadi successivi ci permetterà di compiere l'azione adeguata.

L'altra parte che descriviamo è quella relativa agli statement, ogni riga del corpo di una funzione o del main è necessariamente uno dei costrutti elencati di seguito e , in base al tipo, ne viene costruito l'albero relativo.



Una volta che il codice sorgente Simpla da parsare in ingresso è finito allora avremo all'interno dell'interprete una struttura ad albero che ne descrive la composizione sintattica (eccezion fatta nel caso in cui ci sia un errore sintattico come vedremo dopo).

Cattura e gestione degli errori

Se in qualsiasi momento durante il parsing l'analizzatore lessicale restituisce un simbolo che non è compatibile con la produzione grammaticale che l'analizzatore sintattico si aspetta dovrebbe essere usata allora viene lanciato un errore.

La routine di errore prevede la comunicazione che è avvenuto un errore, la riga nella quale è avvenuto (il numero di linea viene preso direttamente dall'analizzatore lessicale siccome ricordiamo che queste due sezioni lavorano asincronicamente ma in parallelo) e il simbolo che ha causato questa uscita.

Analizzatore semantico

Come detto nella teoria e verificato al passaggio precedente arrivati a questo punto abbiamo a disposizione un albero sintattico astratto.

Questa struttura permette di descrivere completamente il programma in ingresso ma per poterla esplorare correttamente è necessario conoscere le regole con la quale è stata costruita, ovvero il metodo esplorativo non è sempre uguale indifferentemente dal linguaggio implementato o dalle caratteristiche di interprete/compiler.

Questo passaggio non fa ausilio di alcun tool esterno ma è stato scritto direttamente in C.

L'analizzatore semantico del progetto si occuperà del controllo della correttezza semantica del programma in entrata e del type-checking.

Il linguaggio Simpla abbiamo visto essere fortemente tipizzato questo permette di avere un type-checking esaustivo.

Cenni di hashtable

Una hashtable è una struttura che permette di associare ad un insieme di chiavi un insieme di valori, ma soprattutto data una chiave restituisce un valore in un tempo costante (caso medio, nel caso peggiore la cosa può un attimo allungarsi come vedremo dopo).

Per mettere in prospettiva la stessa ricerca (data chiave ottieni valore) fatta tramite un array avrebbe (utilizzando la ricerca binaria) complessità $O(\log n)$ mentre con un'hashtable si parla di $O(1)$ ovvero che io abbia 10 entrate o 1000 la cosa è completamente indifferente in termini di tempo.

Tutto si basa sul fatto che data una chiave esiste un qualche modo per ottenere la posizione in cui quella chiave deve essere nella tabella, ottenuto questo indice la lettura è una semplice operazione di lettura casuale di array.

Il fuoco del discorso è quindi come abbinare ad ogni possibile chiave una posizione univoca, e la risposta è in generale non si può.

Prendiamo come esempio l'insieme delle possibili chiavi l'alfabeto, sappiamo il numero di lettere essere 26, creiamo l'hashtable come un array di dimensione fissa (26) e usiamo come funzione che data una chiave restituisce un indice la semplice conversione della stessa nel suo numero ASCII a cui sottraiamo la costante che permette di abbinare alla prima lettera 0 ($a = 26 - 26$, $b = 27 - 26$..).

La funzione è chiaramente iniettiva, date due chiavi diverse l'indice ottenuto è sicuramente diverso.

Le prestazioni sono come desiderato costanti e pari a quelle di lettura/scrittura casuale di array.

Il problema nasce con l'aumentare della complessità delle chiavi.

Pensando ad esempio a chiavi composte da otto lettere potremmo inserire tutti e otto i caratteri in un intero a 64 bit (8 bit per carattere), convertire la stringa in numero e usarlo come indice dell'array.

Questo richiederebbe di allocare un array di 295,148 petabyte [6] ma anche supponendo la cosa non fosse un problema sarebbe comunque uno spreco siccome praticamente tutte le caselle sarebbero vuote.

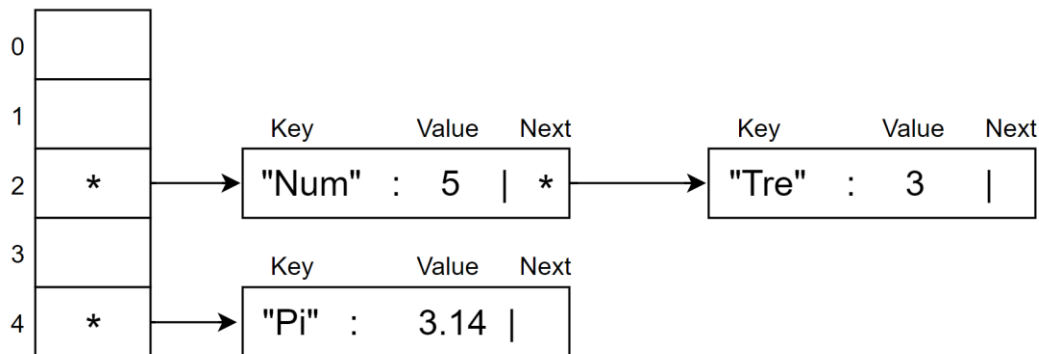
L'idea è quindi che basti un array con una dimensione adeguata per il numero di entry con cui si pensa di avere a che fare e aggiungere un "paracadute" nel caso questa dimensione venga superata.

Il primo problema da superare è che se la dimensione non copre più tutte le casistiche della chiave la funzione precedentemente descritta che converte chiave in indice non è più valida.

La soluzione è l'utilizzo di una funzione di hashing che dato in ingresso una stringa di dimensione arbitraria restituisce un numero in un range prestabilito. Chiaramente per quanto bene la funzione sia stata studiata esistono due stringhe diverse che danno come risultato lo stesso numero, questa situazione è chiamata collisione.

Il metodo usato per gestire le collisioni nel progetto è il separate chaining, ovvero invece che contenere un valore abbinato ad ogni chiave abbiamo un insieme degli stessi.

Nell'implementazione questo si traduce nell'abbinare ad ogni indice una linked-list di valori che verrà fatta scorrere fino a trovare l'elemento interessato nella lettura o fino a trovare l'ultimo elemento nell'inserimento.



Il caso peggiore è quando abbiamo solo una linked-list e dobbiamo percorrerla tutta per trovare il nostro valore.

Implementazione

Strutture

La struttura più importante introdotta è la symbol table, che conterrà tutte le informazioni sulle variabili e funzioni per permettere il type-checking e lo scope delle stesse per controlli semantici.

Symbol table

La symbol table è stata implementata come una tabella con dei campi che vengono definiti chiave e che per ogni istanza della stessa devono essere univoci. In pratica viene scelto il campo nome come chiave primaria, in questo modo se in una certa tabella esiste già una entry con un certo nome non se ne può aggiungere un'altra.

Una struttura di questo tipo può essere implementata tramite una hashtable.

La tabella non è unica, questo perché a scope diversi può corrispondere un diverso numero e tipo di variabili (variabili locali delle funzioni ad esempio).

Esempio di symbol table Simpla

Nome	Classe	Oid	Pointer	nStar	Tipo	Ambiente	nFormali	pFormali	Next
Num	Var	0	0	0	Integer	"Globale"	0	0	NULL

Abbinata ad una variabile globale definita come :

Num : Integer;

Nome : è il nome della variabile/funzione.

Classe : descrive che tipo di entry stiamo descrivendo, i possibili valori sono Var, Par, Fun.

In ordine il significato è : variabile locale o globale dipende dallo scope della tabella stessa, parametro è sempre una variabile ma non viene dichiarata con i metodi convenzionali ma nelle parentesi della firma di una funzione e infine funzione.

Oid : è semplicemente un numero univoco crescente che si abbina ad ogni entrata.

Pointer : è un valore booleano se la variabile/parametro è un puntatore.

nStar : se l'entry è un puntatore qui indichiamo il numero di '*' stars che ne precedono il nome.

Tipo : utile per il type-checking, memorizza il tipo con qui è dichiarata la entry (Integer, Real, String, Boolean)

Ambiente : campo usato solo dalle entry funzione, è un puntatore all'ambiente della funzione stessa, ovvero punta ad un'altra tabella che descriverà il contenuto locale.

nFormali : campo usato solo dalle entry funzione, è il numero di parametri formali richiesti.

pFormali : campo usato solo dalle entry funzione, è un array di dimensione pari a nFormali e ogni elemento è un puntatore ad una entry della tabella puntata da

Ambiente. L'entry puntata è, in particolare, quella di tipo 'Par' che si riferisce al parametro formale della funzione.

Next : puntatore all'eventuale prossimo valore della linked-list, serve per risolvere le collisioni come visto nei cenni di hashtable.

Come già detto questa tabella viene implementata tramite una hashtable, vediamo le caratteristiche principali.

Creiamo la tabella tramite una struct a cui assegnamo un nome (lo scope) e un array di puntatori ad entry di dimensione fissa 1013.

Questo numero è primo per ragioni riguardanti la funzione di hashing, in breve ne aumenta l'efficacia di distribuzione statistica e quindi diminuisce le collisioni.

Sono stati poi implementati dei metodi per l'aggiunta e la ricerca di entry di una tabella.

Una funzione che data una stringa restituisce un indice, la formula con cui questo numero viene computato è molto semplice :

$$h = ((h \ll \text{SHIFT}) + \text{id}[i]) \% \text{TOT};$$
 Con h intero e inizialmente uguale a 0.

Questa istruzione viene ripetuta per ogni lettera della stringa.

I passi sono : prendiamo il valore di h, ne facciamo lo shift logico sinistro di SHIFT (che nel programma è una costante pari a 4) aggiungiamo il valore ASCII della lettera attiva in quel ciclo e infine ne calcoliamo il modulo di TOT (1013 come visto prima).

La prima parte (nelle parentesi) computa un numero che dovrebbe avere una distribuzione statistica abbastanza uniforme, il secondo assicura che il numero computato rientri nella dimensione dell'array.

Type-checking e controllo semantico

Le azioni di controllo tipo e controllo semantico vengono fatte contemporaneamente perché sarebbe uno spreco visitare completamente due volte l'albero per differenziarle.

Prima di vedere quali controlli sono stati eseguiti introduco il funzionamento che permette di ricavare data una qualsiasi espressione il suo tipo, questo è richiesto per verificare la compatibilità di espressioni e sottoespressioni in vari punti dei costrutti.

Come esposto nella sezione di descrizione del linguaggio Simpla le espressioni sono un susseguirsi di uno o più operandi con operatori che li collegano.

Tenendo questo in mente esploriamo la funzione che dato in input la radice di una `expr` o `expr_list` del linguaggio ne restituisce il tipo :

```
type expr(Pnode n, Table *table);
```

Ha due parametri in ingresso, il `Pnode` che conterrà la radice dell'espressione di cui vogliamo ricavare il tipo e la `Table` che ricordiamo essere la symbol table contenente i tipi delle variabili di un certo scope.

La `Table` è chiaramente necessaria perché uno dei possibili fattori è una variabile, e se incontrata il programma deve aver modo di sapere che tipo di valore contiene.

Comunque una volta chiamata inizia l'esplorazione dell'albero a partire da `n` (qui non ripeto tutte le possibili costruzioni che l'albero può avere, sono descritte esaustivamente nella sezione dell'analisi sintattica) scendendo in base all'operatore che incontra chiama la funzione che lo gestisce, questa riceve in ingresso la radice (che conterrà l'operatore che lei sa gestire) e due/un figli/o (operatore binario/unario) chiamerà ricorsivamente sui figli la funzione `expr()` perché un operando può essere una sottoespressione senza limiti di profondità. Quello sopra descritto è in qualche modo il passo ricorsivo, mentre la condizione di terminazione si incontra quando ad `expr()` viene passato un nodo terminale, in quel caso si scende fino alla funzione `factor()` che riconosce il tipo del fattore terminale e lo restituisce con `return`, si risale la pila fino a ritornare al chiamante originale il tipo ottenuto dall'intera espressione (se tutti i passaggi intermedi sono andati a buon fine e i tipi erano compatibili).

Elenchiamo i controlli con una piccola descrizione di come sia stato implementato se non è banale :

- Visibilità degli identificatori referenziati (locali o globali) : facilmente ottenibile avendo la symbol table di ogni scope, il tutto si riduce ad una chiamata di funzione che data la chiave restituisce il valore o `NULL` nel caso sia inesistente.
- Nelle espressioni, compatibilità degli operatori con gli operandi.
Nella spiegazione di funzionamento della funzione `expr()` è stato anticipato

che questa in base all'operatore dell'espressione chiama la funzione adeguata per gestirlo, il motivo per cui non gestisce tutto lei ma delega è proprio perché ogni operatore richiede certi controlli e le funzioni specifiche li conoscono e li mettono in pratica.

- Nell'assegnamento, compatibilità della variabile (LHS) con l'espressione di assegnamento(RHS).

Banale chiamando la funzione `expr()`.

- Tipo 'void' applicabile solo alle funzioni.
- Nella funzione, uguaglianza in numero e tipo dei parametri formali con i parametri attuali.
- Nella funzione, uguaglianza del tipo della espressione di ritorno con il tipo del valore di ritorno.
- Se funzione senza valore di ritorno (tipo 'void') eventuali return privi di argomento.
- Istruzione break situata nel corpo di un ciclo.

Il meccanismo di funzionamento è abbastanza interessante : esiste una variabile booleana `isCiclo` che viene posta ad 1 quando è stato riconosciuto un ciclo e ne sto controllando il corpo, quando incontro un break controllo se questa variabile è vera, in caso contrario non è all'interno di un ciclo.

Il problema principale è che i cicli possono essere nidificati senza limiti di profondità e quindi se avessi un ciclo dentro un ciclo, finito il controllo del corpo di quello più interno la variabile `isCiclo` verrebbe a logica posta a 0 ma così facendo il resto del controllo del ciclo più esterno lancerebbe errore se incontrasse un break, comportamento non corretto.

Per risolvere il problema semplicemente basta controllare se `isCiclo` è già ad 1 quando si incontra un ciclo significa che quest'ultimo è annidato e in chiusura non devo porre la variabile di controllo a 0.

- Compatibilità delle espressioni con le istruzioni in cui sono coinvolte.
- Nelle funzioni non 'void' qualsiasi flow d'esecuzione deve contenere un'istruzione di ritorno.

Viene controllato che non possa succedere per nessuna combinazione delle condizioni booleane di if che si giunga alla fine del corpo di una funzione senza avere un 'return'.

Per farlo usiamo un contatore, se troviamo un'istruzione di ritorno fuori da corpi dell'if la aumentiamo di 1, se invece troviamo un 'return' dentro un corpo dell'if controlliamo se c'è anche nell'altro allora aumentiamo di 1 sennò non viene modificata.

Procedendo in questo modo se la variabile contatore è almeno 1 significa che qualsiasi percorso venga scelto viene comunque incontrata un'istruzione di ritorno come da specifica.

- Nell'assegnamento dell'indirizzo dei puntatori, controllo che il livello della LHS e della RHS siano compatibili.

In Simpla quando si lavora con i puntatori e in particolare nell'assegnamento abbiamo due casistiche possibili :

La variabile da assegnare e la variabile assegnata sono dello stesso livello, in questo caso la sintassi è 'pointer1 = pointer2'.

La variabile da assegnare è un livello più bassa della variabile assegnata, in questo caso la sintassi è 'pointer1 = &pointer2'.

Nella semantica e in particolare in questo passaggio dobbiamo assicurarci che i livelli siano compatibili con la sintassi del codice sorgente.

Cattura e gestione degli errori

Per come funziona questa sezione (esplorazione dell'albero astratto) avremo sempre un nodo che sta venendo controllato, è qui che si capisce l'utilità del campo intero linea all'interno della struttura nodo.

Infatti avendo a disposizione questa informazione quando viene rilevato un errore , che avviene grazie ai controlli dei metodi specifici chiamati a seconda del tipo del nodo attivo, la comunicazione all'utente risulta molto completa potendo includere sia il controllo che non è stato superato sia la linea che contiene lo statement con l'errore.

In ogni caso quando rilevato viene chiamata una procedura che stampa l'errore sullo standard output e termina l'esecuzione del programma.

Fase di esecuzione : Interpretazione

È da questo punto in poi che il progetto, che è un interprete, e un compilatore differiscono.

Questo perché in questa fase iniziamo l'esecuzione del codice ricevuto in input, cosa che in un compilatore avviene successivamente e in un programma separato, avremmo nel suo caso una generazione di codice intermedio che sarebbe poi quello eseguito a monte e che riceverebbe gli input utente.

Ma tornando al progetto, che abbiamo appurato essere un interprete, quello che l'ultima fase ci lascia in input è un albero astratto che rispecchia completamente quello che è il codice Simpla sorgente e che è sintatticamente e semanticamente esatto. Rimane soltanto da eseguirlo con gli eventuali input ricevuti e restituire l'output generato (se c'è).

Tornando per un attimo a parlare dell'architettura del progetto, abbiamo osservato prima che il motivo per il quale type-checking e controllo semantico venissero eseguite insieme fosse per una questione di prestazioni, entrambe le procedure avevano infatti bisogno dell'attraversamento dell'albero, cosa che potevamo evitare di ripetere raggruppandole.

Allo stesso modo però la fase di interpretazione fa uso dell'esplorazione dell'albero per funzionare, quindi perché non unirla alle precedenti? I motivi son due :

1. Prima e forse più importante ragione è che tenendo separati il mondo di analisi (parte comune ad un compilatore) con il mondo di esecuzione permette in un qualsiasi secondo momento di staccare i due per riutilizzarli magari nella creazione di un compilatore o per sostituire l'attuale modo interpretativo con uno completamente diverso senza alcuna difficoltà maggiore di quella che sarebbe copiare/eliminare dei file in formato C.
2. Secondo e comunque di una certa rilevanza è che questa parte è sostanzialmente più lunga e complessa delle due che sono comprese nell'analisi semantica e unire il tutto insieme renderebbe il codice meno comprensibile e manutenibile.

Cenni di stack-based runtime environment

Vengono definiti linguaggi di questo tipo quelli in cui è possibile chiamare una funzione più volte prima che la precedente istanza della stessa sia terminata. In altre parole esistono in esecuzione più istanze della stessa funzione.

Normalmente a run-time si utilizza una struttura esterna chiamata stack che memorizzi il valore attuale delle variabili sia globali che per ogni funzione chiamata. Il blocco memorizzato contenente tutte le informazioni riguardanti una certa istanza di una funzione è detto record di attivazione.

In linguaggi nei quali non esiste la ricorsione, e quindi la chiamata ad una funzione deve terminare prima di permettere una nuova chiamata, lo stack può essere di dimensione fissa perché basta predisporre uno spazio di allocazione per ogni possibile metodo chiamato e la cosa è fatta.

Oltre a ciò decidendo chi viene memorizzato dove durante la creazione dell'interprete è possibile sapere prima dell'esecuzione l'indirizzo delle variabili globali e locali semplificando notevolmente operazioni che spiegheremo successivamente.

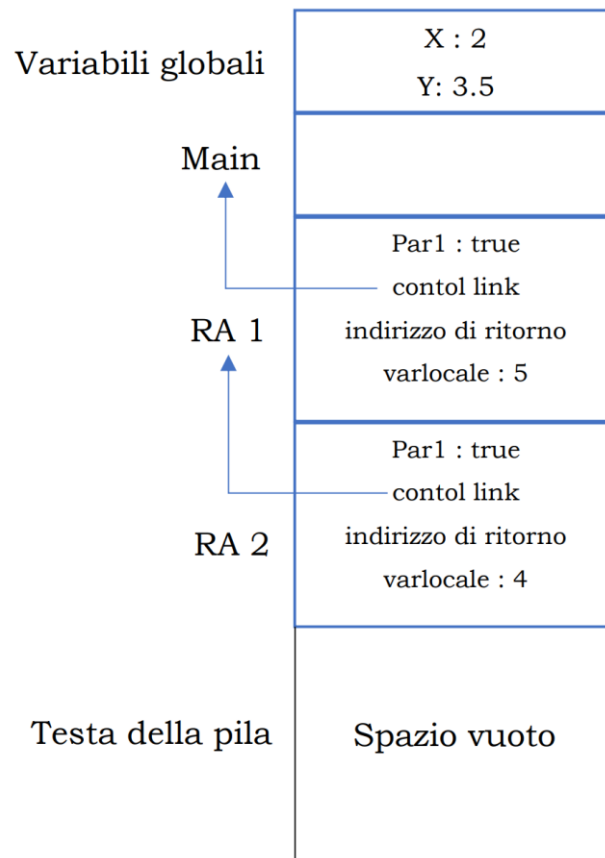
Nessuna delle due cose è però replicabile in un contesto che ammette la ricorsione, in quest'ultimo caso infatti il comportamento e quindi il contenuto dello stack è completamente dipendente dal comportamento del programma e non può essere previsto. È questo il caso di Simpla che prende il nome di stack-based runtime environment.

L'idea è quella di aggiungere e rimuovere dinamicamente i record di attivazione nel momento in cui viene chiamata o terminata una funzione.

Nella pratica si utilizza uno stack come detto, le funzioni push e pop riproducono il comportamento voluto, il blocco in cima alla pila sarà quello attivo in quel momento.

Prima di continuare nella spiegazione c'è un'ulteriore distinzione da fare, esistono linguaggi che permettono la definizione di una funzione all'interno di un'altra funzione (ambienti stack-based con procedure locali) oppure che non lo permettono (senza procedure locali). Simpla è del secondo tipo e la spiegazione teorica qui in poi si rifà a quella casistica.

Principalmente un record di attivazione contiene : parametri della funzione, variabili locali e temporanee, un puntatore al RA precedente (quello che ha effettuato la chiamata) e l'indirizzo dal quale riprendere l'esecuzione nel record di attivazione precedente.



Per le motivazioni sopra esposte non è più possibile avvalersi di indirizzi statici per utilizzare variabili/parametri, introduciamo il concetto di offset.

Ipotizzando la composizione del blocco RA come nel disegno sopra avremo in ordine : parametri, control link, indirizzo di ritorno, variabili locali.

Sapendo che la composizione è sempre così formata e avendo un puntatore al record corrente è facilmente ottenibile qualsiasi campo (sapendo la dimensione di ogni componente) semplicemente scorrendo i bytes.

Proprio per questa ragione esiste un puntatore che memorizza sempre l'indirizzo del RA corrente.

Implementazione

Strutture

Stack di attivazione e stack degli oggetti

Nella pratica abbiamo due stack :

- Stack degli oggetti che è una struttura a due campi : tipo (integer, real ...) valore di tipo Value (union vista nella sezione Lessicale).
- Stack di attivazione, anch'essa una struttura con campi : numero di oggetti di tipo intero, un puntatore di tipo Stack degli oggetti, un puntatore a Symbol table.

Lo stack di attivazione tiene traccia dei record di attivazione ancora sulla pila, ogni elemento tiene traccia del numero di variabili/parametri che gli appartengono e sono sullo stack degli oggetti, punta al primo di questi oggetti e punta alla Symbol table appartenente alla funzione di cui lui è istanza.

Lo stack degli oggetti invece semplicemente tiene traccia di tutti gli oggetti ovvero coppia di valore e tipo che sono attivi nel programma in quel momento.

Stack di attivazione

N oggetti	*oggetto	*Symbol
3		*

Astack

Stack degli oggetti

Tipo	Valore
Integer	5
String	"Vuoto"

Ostack

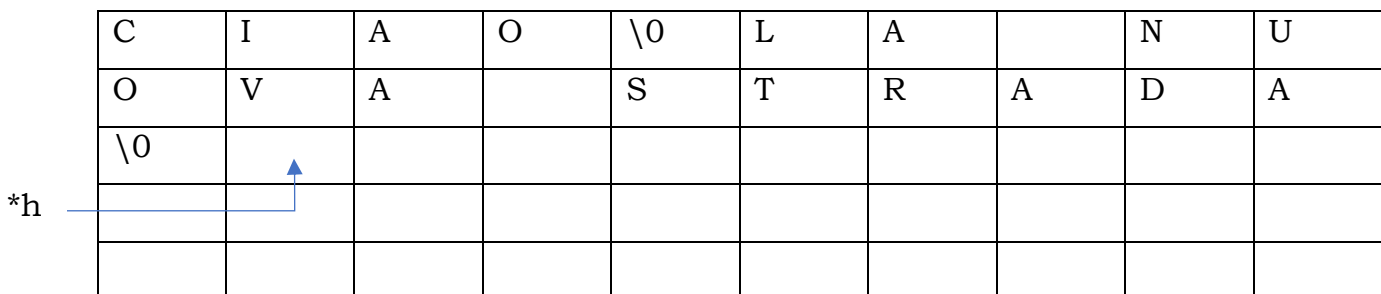
'ap' e 'op' sono due variabili che puntano sempre alla prima casella dei rispettivi stacks.

String pool

Riprendendo per un attimo la struttura stack degli oggetti vediamo che il valore viene memorizzato in un tipo Value, che andandone a riprendere la definizione ci accorgiamo essere una 'union' con possibili valori 'integer', 'real', 'boolean', 'string'. In particolare però 'string' è definito come puntatore, quindi in realtà sulla pila viene memorizzato solo un indirizzo dove potremo trovare l'effettivo valore della frase.

La struttura che contiene le stringhe, ovvero tipi di dati di dimensione variabile, è la String pool.

La funzionalità che deve garantire è di permettere di aggiungere una qualsiasi stringa e restituirne il puntatore se questa è la prima volta che viene aggiunta, mentre se esiste già una frase identica in memoria allora deve garantire di poterla trovare e restituirne il puntatore senza utilizzare ulteriore memoria.



C	I	A	O	\0	L	A		N	U
O	V	A		S	T	R	A	D	A
\0									

Nel codice C riprodurremo il comportamento della memoria tramite un array, che verrà chiamato heap (seguendo i linguaggi già esistenti).

Guardando la tabella sopra (che raffigura l'array) vediamo che ogni casella contiene una lettera, ovvero è un byte, e per riconoscere quando una frase termina si sfrutta il meccanismo già usato in C, appena riconosciuto il carattere di fine stringa '\0' allora terminiamo.

Il puntatore h (che vedremo dopo in funzione) serve per tenere sempre traccia della prima casella disponibile libera.

L'heap funziona perfettamente per aggiungere stringhe in maniera permanente e con dimensione variabile, però non risolve il problema del tenere traccia di quali

sono già aggiunte e se dovessimo ogni volta controllare il contenuto le prestazioni sarebbero pesantemente intaccate.

Per questo motivo seguendo leggermente l'implementazione di java sfruttiamo una Hash table, struttura già vista nella Symbol table.

In pratica si utilizza come chiave la frase che si vuole aggiungere e come valore associato ad essa il puntatore allo heap.

Esecuzione

Si parte dalla radice dell'albero sintattico astratto, per prima cosa creo il RA del main, sintatticamente il body della funzione main non può essere vuoto quindi questa operazione è sicuramente da fare.

Aggiungo le variabili globali allo stack degli oggetti, queste saranno collegate allo stack di attivazione in cima alla pila in quel momento e cioè il main.

Facciamo un piccolo esempio con codice Simpla

```
Num : integer;  
Pi : real;  
body  
    Num = 3;  
    Pi = 3.14;  
end.
```

Stack di attivazione

N oggetti	*oggetto	*Symbol
2		

Astack

Stack degli oggetti

Tipo	Valore
Integer	#UNDEF
Real	#UNDEF

Ostack

Symbol table Globale

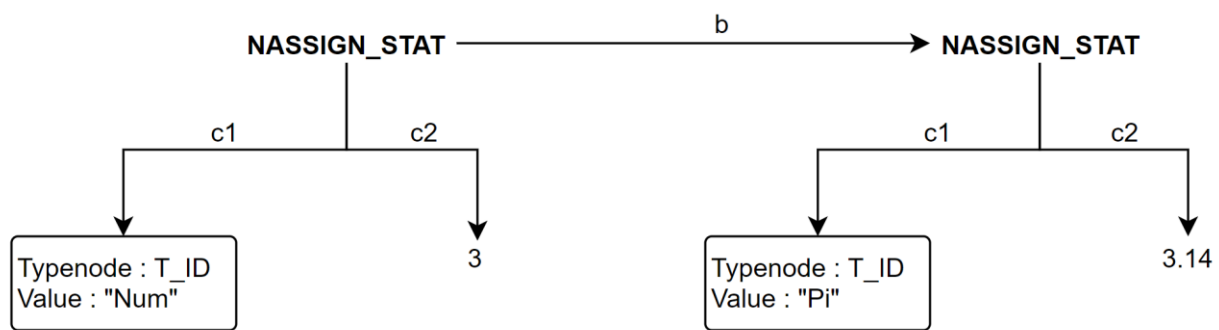
Nome	Classe	Oid	Pointer	nStar	Tipo	Ambiente	nFormali	pFormali	Next
Num	Var	0	0	0	Integer	"Globale"	0	0	NULL
Pi	Var	1	0	0	Real	"Globale"	0	0	NULL

Al momento i valori nella pila sono stati segnati come Undefined, in realtà un valore ci sarebbe ma è imprevedibile e derivante dalla lettura di una casella 'sporca' della memoria.

A questo punto inizia l'esecuzione del corpo principale, per come è definito Simpla è stato possibile creare un unico metodo che gestisce l'esecuzione del corpo del main, delle funzioni e di tutti i costrutti che ne hanno uno (if, for, while).

In sostanza ogni riga del body sarà uno statement e il metodo si occupa tramite il contenuto informativo del metodo (come spiegato prima gli enum Typenode, Nonterminal) di chiamare il metodo corretto per la gestione di quella riga per poi riprendere il controllo alla fine di quella.

Ad esempio riprendendo il codice precedente l'albero del body sarebbe così costruito :



Guardiamo brevemente il comportamento interno con i vari possibili statement con cui l'interprete può trovarsi a lavorare.

Assegnamento

Incontriamo subito in questo costrutto una routine che si ripeterà spesso in questo capitolo, ovvero come ricavare la posizione sullo stack di una variabile dato il nome.

Il primo passaggio consisterà nel ricavare l'object identifier (oid) della variabile richiesta, questo numero è presente nella Symbol table dell'ambiente in cui essa è dichiarata e ricordiamo essere un intero univoco all'interno della tabella.

Questo numero indicherà l'offset, dall'inizio del record di attivazione, richiesto per trovare la casella corretta sullo stack degli oggetti.

Vediamo un piccolo esempio

Stack di attivazione

N oggetti	*oggetto	*Symbol
2		*
2		

Astack

Stack degli oggetti

Tipo	Valore
Integer	13
Real	5.55
String	"pi"
Integer	55

Ostack

Symbol table calcPi

Nome	Classe	Oid	Pointer	nStar	Tipo	Ambiente	nFormali	pFormali	Next
Nome	Var	0	0	0	String	"calcPi"	0	0	NULL
nVolte	Var	1	0	0	Integer	"calcPi"	0	0	NULL

Le strutture interne sopra riportate descrivono una situazione in cui abbiamo sullo stack di attivazione due record : main e uno che sappiamo appartenere alla funzione 'calcPi' dal nome dell'ambiente della Symbol table.

Sullo stack degli oggetti abbiamo quattro variabili, due apparterranno al primo RA due al secondo (lo si evince dai puntatori).

In questa situazione applicando la routine di prima e volendo ad esempio ricavare la posizione di 'nVolte' procederemo in questo modo :

Otteniamo la Symbol table dal RA attivo in quel momento, chiediamo ad essa l'oid della variabile con il nome che ci interessa, verrà restituito '1' in questo caso.

A questo punto partendo dalla casella dello stack degli oggetti puntata dal RA attivo ('String : "pi") lo scorriamo per un numero di posizioni pari all'oid (1).

Ciò che otteniamo è effettivamente la casella che ci interessa ovvero quella contenente il valore che vogliamo leggere/modificare.

Il tutto naturalmente funziona perché l'interprete in fase di aggiunta allo stack delle variabili dichiarate nel programma si assicura che venga mantenuto l'ordine della Symbol table.

Tornando al costrutto che stavamo descrivendo, la prima cosa che viene fatta è risolvere l'espressione della RHS, il valore verrà posto in cima allo stack degli oggetti.

Ora viene ricavata la posizione della variabile della LHS dello statement e viene assegnata ad essa il valore in cima allo stack.

Una volta finito viene eliminato il primo elemento dello stack per ripulire gli elementi non più utili.

If

Viene risolta l'espressione condizionale e il valore memorizzato in cima alla pila.

Si legge il primo valore dello stack, se 'true' viene eseguito il corpo dell'if, se 'false' viene, se esistente, eseguito il corpo dell'else.

For

Ricordo che a differenza del C nel Simpla non è possibile specificare l'incremento della variabile che stiamo iterando.

Per prima cosa viene eseguita l'azione che definiamo una tantum che solitamente si presenta come il classico assegnamento $i=0$.

In Simpla questo statement dev'essere necessariamente un assegnamento, verrà gestito come spiegato nel paragrafo dedicato all'assegnazione.

Dopodiché si entra in un ciclo, la condizione di permanenza viene controllata all'inizio dello stesso in maniera simile al costrutto if.

Risolve l'expr condizionale di permanenza, il suo valore verrà posto in cima allo stack, lo prelevo lo controllo e se positivo continuo.

Viene chiamata la funzione che gestisce i corpi dei costrutti, quando il controllo viene restituito possono essere accadute due cose : le istruzioni del corpo sono state tutte eseguite e devo riiniziare, è stato incontrato un break e in questo caso tramite una variabile globale me ne accorgo e termino.

Se la terminazione non è prematura l'ultimo passaggio è l'incremento della variabile usata per l'iterazione e infine si riinizia.

Per come esposto il comportamento si potrebbe notare che la variabile di supporto viene aumentata sempre prima di essere controllata e quindi finito il for a meno di presenza di istruzioni 'break' avrà sempre un valore di una unità superiore al limite che avevamo posto nel codice.

Un dettaglio non troppo influente ma da tenere a mente e volendo modificabile facilmente.

While

Il funzionamento è identico al 'for' tranne l'istruzione iniziale che viene eseguita solo una volta, l'assegnamento che qui non è presente.

Chiamata di funzione

Per prima cosa (solo se la funzione di ritorno è di tipo diverso da 'void') creiamo uno spazio sullo stack degli oggetti che ospiterà il valore di ritorno (viene fatto semplicemente spostando il puntatore op, che ricordiamo punta sempre alla prima casella disponibile sulla pila, e ponendo il tipo della casella uguale a quello della funzione).

Nel momento in cui viene fatto siamo ancora sull'RA vecchio, questo perché questo valore vogliamo rimanga anche quando disallocheremo la funzione chiamata dallo stack di attivazione e conseguentemente da quello degli oggetti.

Stack di attivazione

N oggetti	*oggetto	*Symbol
2		*
3		*

Astack

Stack degli oggetti

Tipo	Valore
Integer	13
Real	5.55
String	"pi"
Integer	55
Boolean	#UNDEF

Ostack

ap

op

I passaggi per arrivare alla situazione sopra sono i seguenti: il primo RA appartiene sicuramente al main, il secondo sarà di qualche altra funzione di cui potremmo sapere il nome controllando la Symbol table associata.

Infine siamo ad un punto in cui la funzione sta chiamando a sua volta un'altra funzione, viene predisposta la casella sullo stack degli oggetti come già detto e lo vediamo dalla casella #UNDEF, il valore verrà assegnato alla fine dell'esecuzione.

C'è un ultimo passaggio che dobbiamo compiere prima di creare il nuovo RA, ovvero calcolare il valore degli eventuali argomenti passati alla funzione.

Deve essere fatto in questo momento perché un argomento può essere anche una variabile e in questo caso per trovarla senza meccanismi particolari devo avere attivo il suo RA.

Spiegato più semplicemente se voglio sapere il valore di una variabile non posso controllare tutto lo stack degli oggetti perché posso trovarne più di una o soltanto una ma inerente ad un RA che non ha nessun collegamento con quello in cui stiamo lavorando in quel momento, quindi per procedere in maniera standard devo trovarmi nel RA del quale la variabile dovrebbe appartenere (sia essa locale o globale).

Possiamo ora creare il RA inerente alla funzione chiamata e che sapremo essere di tipo Boolean guardando la figura sopra (la casella riservata per il suo valore di ritorno è di quel tipo).

Ipotizzando due parametri la situazione a questo punto sarebbe :

Stack di attivazione

Stack degli oggetti

N oggetti	*oggetto	*Symbol
2		*
3		*
2		*

Astack

Tipo	Valore
Integer	13
Real	5.55
String	"pi"
Integer	55
Boolean	#UNDEF
Integer	33
Boolean	true

Ostack

ap

op

Da qui il comportamento è uguale al main visto prima, vengono aggiunte le variabili locali e viene eseguito il corpo.

Quando avrò finito l'esecuzione rimuovo dallo stack di attivazione l'RA e sistemo i puntatori ap, op alla loro posizione precedente.

Vediamo nel prossimo costrutto, il 'return', come venga passato l'eventuale valore passato.

Return

Se il costrutto è seguito da un'espressione la computo chiamando la funzione apposta che lascerà in cima alla pila il valore computato.

Prendo il valore e lo assegno all'ultima casella del RA precedente, la casella era riservata proprio per questo utilizzo come spiegato sopra.

Infine rimuovo il valore in cima alla pila perché non più utile.

La chiamata di questo costrutto ha però anche la finalità di interrompere l'esecuzione del corpo, per renderlo possibile esiste una variabile booleana che viene posta ad 1 quando questo viene chiamato, durante l'esecuzione della funzione che esegue i corpi dei costrutti controllo sempre se questa è vera, se lo è interrompo e ritorno il controllo al chiamante.

Read

Questo costrutto sfrutta lo 'scanf()' del C rendendo quindi il funzionamento banale.

Semplicemente è un assegnamento con RHS letta dallo standard input

Write

Anche qui viene sfruttata una funzionalità del C 'printf()', il valore che vogliamo stampare passa prima per la funzione `expr()` che lo computerà e lo memorizzerà in cima alla pila, il valore verrà prelevato e in base al tipo stampato con la routine corretta.

Expr

Abbiamo fin qui sfruttato in maniera consistente una funzione chiamata 'expr()' senza però definire a questo livello come funzioni.

La sua struttura è identica alla controparte nell'analisi semantica, ricordiamo solo il largo uso della ricorsione e la discesa fino a `factor`.

A differenza di prima però il suo scopo non è controllare ma eseguire le azioni richieste, la cosa è molto facile avvalendosi degli operatori che il C già ha a disposizione, la cosa si riduce a riconoscere l'operatore corretto e usarlo nel codice C.

Un dettaglio degno di nota è il metodo risolutivo per, ad esempio, espressioni molto lunghe.

In questo caso verranno calcolati i vari pezzi ognuno formato da un operatore e un numero di fattori dipendenti dell'operatore stesso seguendo l'ordine definito nelle specifiche, però serve un modo per memorizzare questi vari pezzi fino al calcolo del numero finale.

Per farlo si utilizza lo stack come spazio di memorizzazione temporaneo.

Era già stato illustrato che i RA avevano compreso al loro interno una parte per le variabili temporanee.

Facciamo un esempio con l'espressione "15+2*8/4"

Senza ripetere come viene esplorato l'albero da `expr` in giù (già visto nei capitoli precedenti) sappiamo che il primo nodo che incontriamo ha come operatore il '*', i suoi figli saranno "15+2" e "8/4".

Chiameremo ricorsivamente sui figli la funzione fino a quando sullo stack avremo tutti i fattori e non resterà che applicare gli operatori correttamente :

```
-----OSTACK(TIPO|VAL)-----  
[ INTEGER | 15 ]  
[ INTEGER | 2 ]  
[ INTEGER | 8 ]  
-----ASTACK(nOggetti|Val del primo)-----  
[ 3 | 15 ]
```

```
-----OSTACK(TIPO|VAL)-----  
[ INTEGER | 15 ]  
[ INTEGER | 16 ]  
[ INTEGER | 4 ]  
-----ASTACK(nOggetti|Val del primo)-----  
[ 3 | 15 ]
```

```
-----OSTACK(TIPO|VAL)-----  
[ INTEGER | 15 ]  
[ INTEGER | 4 ]  
-----ASTACK(nOggetti|Val del primo)-----  
[ 2 | 15 ]
```

```
-----OSTACK(TIPO|VAL)-----  
[ INTEGER | 19 ]  
-----ASTACK(nOggetti|Val del primo)-----  
[ 0 | 19 ]
```

Il testo sopra riportato è l'output che l'interprete ci propone quando gli chiediamo di stampare lo stack che ha internamente durante l'esecuzione.

Da qui possiamo vedere come vengano aggiunti i fattori e sommanti nell'ordine di precedenza degli operatori.

La legenda del contenuto tra le parentesi quadre è scritta nel nome dello stack.

Puntatori

I puntatori sono trattati come normali variabili, però a seconda di come vengano scritti hanno diversi significati e valori.

Come in C le due forme possibili sono “*’var per accedere al valore puntato e var per accedere invece all’indirizzo che sta venendo puntato.

Prendiamo come esempio il codice:

```
a,*p1,**p2,**p3,***p4: real;

body
    writeln("num, inutile poi sovrascritto (reale) : ");
    read(a);
    p1 = &a;
    p2 = &p1;
    p3 = p2;
    p4 = &p3;
    ***p4 = 5.0;
    writeln("Chain : ",&p4,"->",p4,"->",*p4,"->",**p4,"->",***p4);
    printStack();
end.
```

L’output che ci si presenta è : “Chain : 4->3->1->0->5.00”

Con stack finale :

```
-----OSTACK(TIPO|VAL)-----
[ REAL | 5.000000 ]
[ *REAL | 0 ]
[ *REAL | 1 ]
[ *REAL | 1 ]
[ *REAL | 3 ]
-----ASTACK(nOggetti|Val del primo)-----
[ 5 | 5.00 ]
```

Proviamo a spiegarlo partendo dal significato del codice.

Abbiamo quattro puntatori tutti reali e una variabile normale, ogni puntatore ha un livello di profondità crescente da uno a tre.

Chiediamo all’utente un numero reale che poi sovrascriveremo, e lo assegnamo

all'unica variabile normale ('a').

Ora tramite l'operatore '&' che restituisce l'indirizzo di una variabile diversamente dal solito valore assegnamo al primo puntatore l'indirizzo di 'a' : `p1 = &a;`

Notiamo che nel caso del puntatore non mettiamo l'operatore '*' prima della variabile perché vogliamo modificare l'indirizzo a cui punta non il valore puntato.

Ora salendo di un livello assegnamo a p2 l'indirizzo di p1.

La situazione fin lì è : p2 punta a p1 che punta ad a (con punta intendo ha come indirizzo puntato quelli di ...).

Ora assegno a p3 che ha un livello di profondità uguale a p2 l'indirizzo puntato da p2, la differenza qui è che p3 non passerà attraverso p2 per ricavare il valore puntato finale perché essi sono entrambi allo stesso livello. Ovvero ci troveremo due catene che puntano alla stessa cosa con percorsi diversi entrambi tramite p1.

Infine assegno a p4 l'indirizzo di p3.

L'output del programma stamperà rispettivamente :

Indirizzo nella memoria di p4 -> l'indirizzo puntato da p4 -> l'indirizzo puntato da p3 ...

I numeri che si vedono in output sono quindi indirizzi della memoria interna dell'interprete e siccome lo stack è costruito come un array i numeri sono interi e partono da 0.

Spiegando velocemente il contenuto dello stack partendo dall'alto :

La prima casella appartiene ad 'a' e contiene il numero ormai modificato perché la stampa è di quando il programma è ormai finito.

Subito sotto troviamo il puntatore p1, esso ha come valore l'indirizzo a cui punta, che è 'a', ovvero 0.

p2 e p3 come detto puntano alla stessa cosa e non si conoscono tra loro (non sono parte della stessa catena) e quindi vedremo hanno lo stesso indirizzo p1 puntato.

Infine p4 conosce p3 e ha in memoria l'indirizzo di quest'ultimo.

Conclusione

Il progetto ha mostrato una possibile implementazione per la creazione di un interprete di un linguaggio di programmazione.

La relazione voleva soprattutto dare enfasi sulla modularità che un programma del genere ha, molte delle parti possono essere sostituite senza che le altre si accorgano minimamente e soprattutto come detto la possibilità di trasformarlo in compilatore (o vice-versa) richiederebbe solo la scrittura di quella parte e potrebbe avvalersi di tutto il lavoro già fatto.

Molte delle scelte fatte internamente sono con mentalità statica, la dimensione dello stack ad esempio abbiamo visto essere fissa il che in ottica di funzionamento reale potrebbe comportare un limite o spreco abbastanza importante. La cosa si sarebbe potuta risolvere in maniera anche piuttosto facile ma non era lo scopo del programma e i problemi che porta non inficiano la volontà puramente didattica dell'implementazione.

Appendice

Grammatica BNF

program \rightarrow var-decl-list func-decl-list body .
var-decl-list \rightarrow var-decl var-decl-list | ϵ
var-decl \rightarrow id-list : type ;
id-list \rightarrow **id** , id-list | **id**
type \rightarrow **integer** | **real** | **string** | **boolean** | **void**
func-decl-list \rightarrow func-decl func-decl-list | ϵ
func-decl \rightarrow **func id** (opt-param-list) : type var-decl-list body ;
opt-param-list \rightarrow param-list | ϵ
param-list \rightarrow param-decl , param-list | param-decl
param-decl \rightarrow **id** : type
body \rightarrow **body** stat-list **end**
stat-list \rightarrow stat ; stat-list | stat ;
stat \rightarrow assign-stat | if-stat | while-stat | for-stat | return-stat | read-stat |
write-stat | func-call | **break**
assign-stat \rightarrow **id** = expr
if-stat \rightarrow **if** expr **then** stat-list opt-else-stat **end**
opt-else-stat \rightarrow **else** stat-list | ϵ
while-stat \rightarrow **while** expr **do** stat-list **end**
for-stat \rightarrow **for id** = expr **to** expr **do** stat-list **end**
return-stat \rightarrow **return** opt-expr
opt-expr \rightarrow expr | ϵ
read-stat \rightarrow **read** (id-list)
write-stat \rightarrow write-op (expr-list)
write-op \rightarrow **write** | **writeln**
expr-list \rightarrow expr , expr-list | expr
expr \rightarrow expr logic-op bool-term | bool-term
logic-op \rightarrow **and** | **or**
bool-term \rightarrow rel-term rel-op rel-term | rel-term

rel-op \rightarrow == | != | > | >= | < | <=

rel-term \rightarrow rel-term low-prec-op low-term | low-term

low-prec-op \rightarrow + | -

low-term \rightarrow low-term high-prec-op factor | factor

high-prec-op \rightarrow * | /

factor \rightarrow unary-op factor | (expr) | **id** | const | func-call | cond-expr | cast (expr) | **addr id**

unary-op \rightarrow - | **not**

const \rightarrow **intconst** | **realconst** | **strconst** | **boolconst**

func-call \rightarrow **id** (opt-expr-list)

opt-expr-list \rightarrow expr-list | ϵ

cond-expr \rightarrow **if** expr **then** expr **else** expr **end**

cast \rightarrow **integer** | **real**

Esempio codice : Fibonacci

```
i, count : integer;

func fibonacci(n : integer) : integer
body
    count = count + 1;
    if (n <= 0) then
        return 0;
    end;

    if (n == 1) then
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
    end;
end;

body
    write("Inserisci un numero: ");
    read(i);

    count = 0;
    writeln("Fibonacci(", i, ") = ", fibonacci(i));
    writeln("Chiamate effettuate: ", count);
end.
```

Esempio codice : Puntatori

```
a,*b : integer;

func somma(a : integer, b : integer, *riporto : integer) : void
temp : integer;
body
```

```
    fun(a, b, &temp);  
    *riporto = temp;  
end;
```

```
func fun(primo : integer, secondo : integer, *addr : integer) : void  
body  
    *addr = primo + secondo;  
end;
```

```
body  
    somma(5, 16, &a);  
    b = &a;  
    writeln(a);  
end.
```

Riferimenti

- [1] A. V. Aho, M. S. Lam, R. Sethi and J. D. Ullman, Compilers: Principles, Techniques, and Tools, 2006.
- [2] T. Reps. [Online]. Available:
<https://research.cs.wisc.edu/wpis/abstracts/toplas98b.abs.html>.
- [3] [Online]. Available: <https://gianfranco-lamperti.unibs.it/tl/lezioni/tl-sintattica.pdf>.
- [4] [Online]. Available: https://en.wikipedia.org/wiki/Regular_expression.
- [5] [Online]. Available: <https://www.epaperpress.com/lexandyacc/prl.html>.
- [6] [Online]. Available: <https://craftinginterpreters.com/hash-tables.html>.