SOMMARIO

INFORMAZIONI GENERALI

- Linguaggio scelto : C.
- Librerie esterne : Nessuna.
- Tipo di programma : Interprete.

File utili :

- Input.txt -> File contenente codice sorgente Simpla.
- Strutture -> File contenente Symbol table (globale + locali), Ostack e Astack a fine esecuzione, Hashtable di tutte le stringhe memorizzate in “heap”, il contenuto dell’heap.

Funzioni utili a fini di debug (se servono per capire qualcosa) :

- void printAStack() (runStructure.h) ovunque chiamata printa il contenuto di Ostack e Astack in quel momento (Nel file Strutture non sullo standard output).

ANALIZZATORE LESSICALE

FILE COINVOLTI

lexer.lex, lexer.c, def.h, parser.h.

STRUTTURE COINVOLTE

Nel file def.h troviamo la definizione della union Value che sarà il tipo che utilizzeremo lungo tutto il programma per contenere dati.

Nel file parser.h troviamo l’enum yytokentype generato da bison e usato come una sorta di vocabolario in comune tra analizzatore lessicale e sintattico.

DESCRIZIONE

Primo passo dell’interprete è l’analisi lessicale del codice sorgente.
Il file lexer.lex contiene le regole poi trasformate in codice c da flex.

DICHIARAZIONI

Definisco un contatore linea che verrà utilizzato in tutte le altre fasi del programma per specificare la posizione dell'errore e una variabile Value utilizzata come deposito di dati poi da passare all'analizzatore sintattico.

REGOLE DI TRADUZIONE

Definisco le regole per matchare tutte le keyword ed elementi significativi del linguaggio Simpla. La regola 'acapo' è scritta in maniera un attimo particolare perché sistemi operativi diversi hanno diversi modi di definire il newline e tendeva a dare problemi, definita così cattura '\n', '\r\n', '\r' l'ultimo dei quali non è standard ne su unix ne su windows ma è considerato possibile e quindi l'ho aggiunto.

Le altre regole sono abbastanza standard e prese per la maggior parte delle slides del corso.
Nel caso in cui un carattere non venga riconosciuto viene comunicato sullo standard output specificando la linea nella quale è presente e chiamando la funzione errLessicale() che provvederà a terminare il programma.

ANALIZZATORE SINTATTICO

FILE COINVOLTI

Parser.y, parser.c, parser.h, def.h

STRUTTURE COINVOLTE

In def.h definiamo due enum (Nonterminal, Typenode) contenenti valori per specificare il tipo di nodo.
Inoltre chiamiamo Node la struttura che useremo poi per ogni nodo, essa contiene un campo Typenode che definisce il tipo di terminale o indica se è un nonterminale e in quest'ultimo caso nel campo value.ival memorizzeremo il tipo Nonterminal che qualifica il nodo. Inoltre troviamo in node un campo intero linen che indica la riga (spiego dopo in descrizione) e tre puntatori c1,c2,b con i quali collegherò il padre ai figli e fratelli.

In parser.h abbiamo la stessa enum utilizzata anche in lex che come abbiamo detto funge da vocabolario in comune ed è quindi definita anche qui.

DESCRIZIONE

Nel progetto il blocco di analisi sintattica, similmente a quello lessicale, si avvale di un tool esterno (bison) per la generazione di codice C, parser.y -> parser.c.

Il file parser.y che contiene le regole lessicali (oltre che il main) è cosi composto :

DICHIARAZIONI BLACK BOX :

Includo diversi file header di cui poi spiego l'utilizzo.

Con la direttiva #define YYSTYPE Pnode faccio override del tipo della variabile yyval da int a Pnode.

Yytext, lexval, line sono variabili di lex contenenti rispettivamente testo matchato, variabile ti tipo Value con uno dei campi contenente il valore letto, il contatore numero linea attuale.

Yyin variabile di tipo FILE alla quale faremo puntare 'Input.txt' così da non dover dare il codice sorgente tramite standard input.

Pnode root contenente il puntatore al primo nodo dell'AST che verrà generato

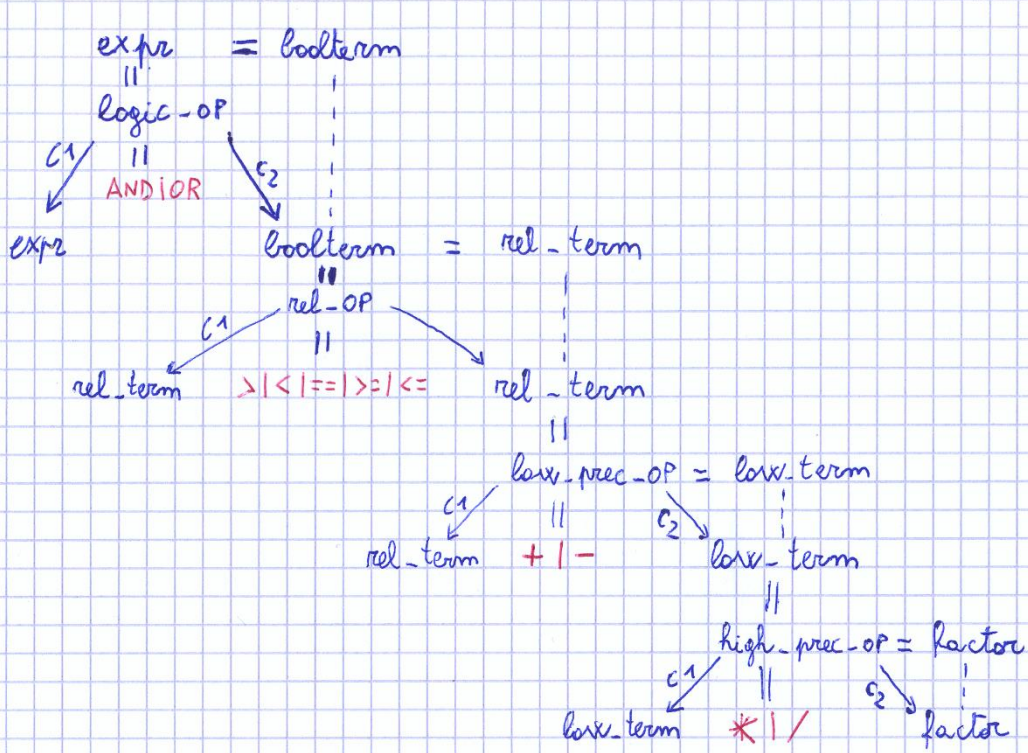
DICHIARAZIONI WHITE BOX :

Contiene tutti i token che rappresentano quello che ho precedentemente definito come il vocabolario in comune, tramite questi bison genererà l'enum yytokentype in parser.h.
Con %start program definisco da quale regola partire per il parsing (superfluo siccome di default è il primo, messo più per chiarezza di lettura).

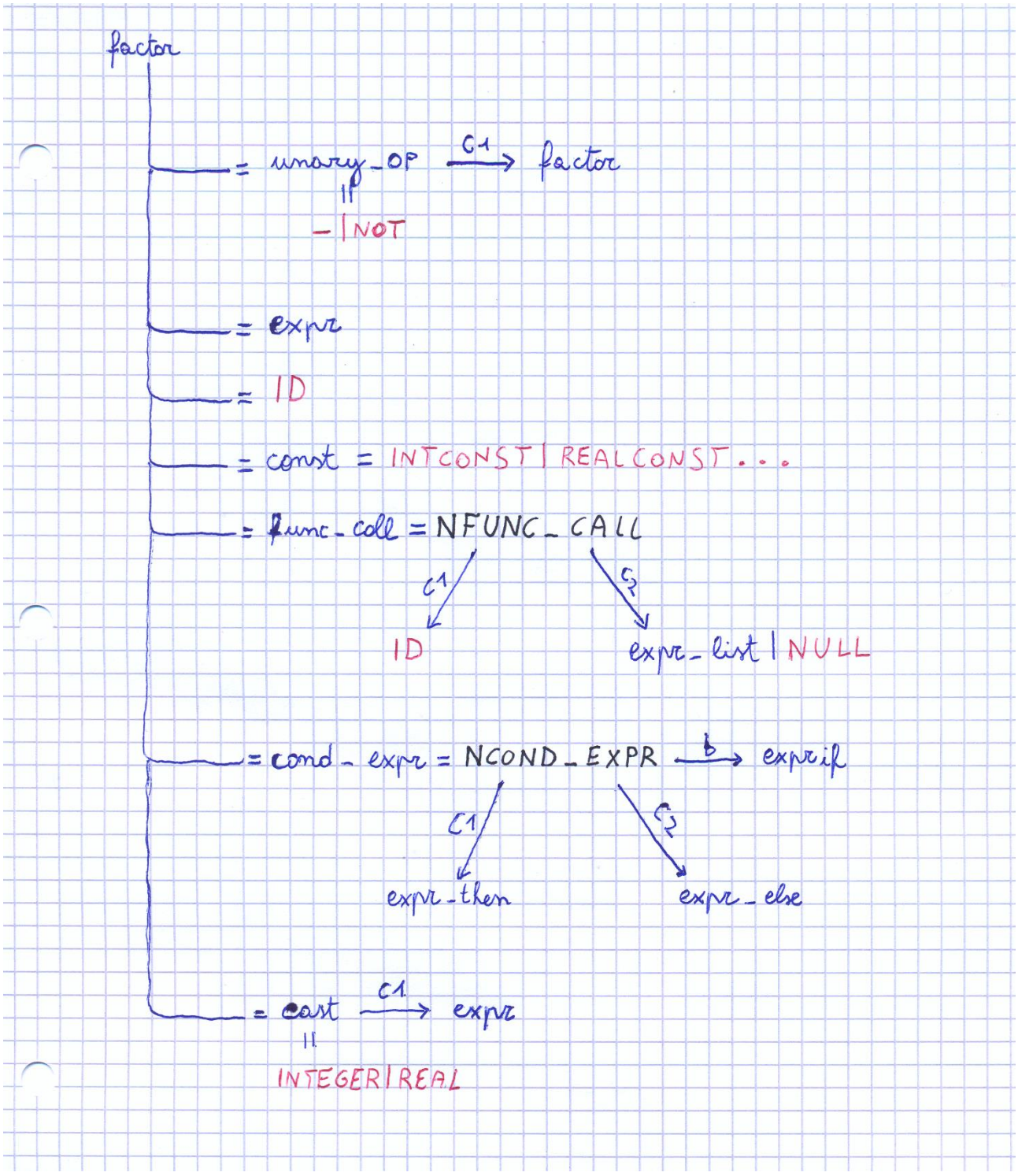
REGOLE DI TRADUZIONE :

Ogni regola ha abbinato (implicitamente o esplicitamente) un suo frammento di codice c corrispondente all'azione semantica.
Credo che una spiegazione sia abbastanza inutile, provo ad inserire qualche immagine di frammenti di come si presenta questo albero genericamente. Aggiungo solo che le azioni semantiche costruiscono quello che è l'AST (abstract syntaxt tree) che è poi lo strumento usato nel resto dell'interprete.

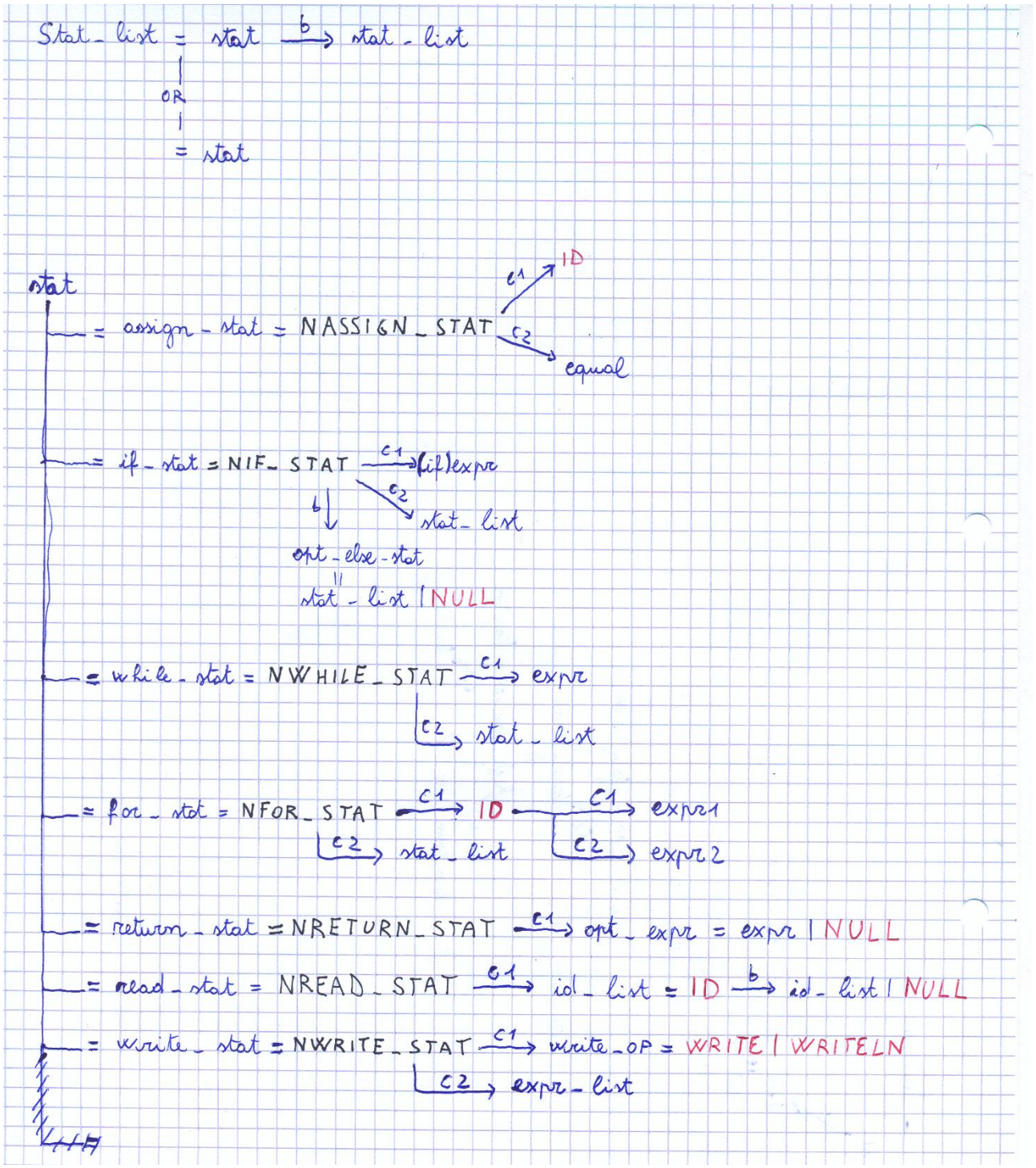
Esploro come prima cosa l'albero da expr in giù che è secondo me la parte più importante.
Nei nodi spesso compaiono due '=' , stanno ad indicare le due possibili alternative, ad esempio da expr se viene matchata la prima regola (abbiamo un 'AND' o 'OR') allora scenderemo per l'uguale logic_op sennò l'altro.
Da notare che a nessuno di questi nodi viene specificato il tipo di Nonterminal come specificato nella grammatica astratta del linguaggio.
Ho segnato in rosso i keynode e in nero i Nonterminal.


expr = boolterm
logic-OP
AND|OR
c1
c2
expr
boolterm = rel-term
rel-OP
>|<|==|>=|<=
rel_term
rel_term
low_prec_OP = low_term
+|-
rel_term
low_term
high_prec_OP = factor
*|/
low_term
factor

Factor similmente a stat presenta un insieme di possibili alternative.



E infine l'albero di Stat-list che è il corpo di tutte le funzioni, compreso il main, e delle strutture di controllo/loop.



(Lascio le immagini nella cartella presentazione se servisse per ingrandirle o altro).

FUNZIONI AUSILIARIE :

La maggior parte dei metodi sono quelli esplorati nelle slides del corso, aggiungo solo il main nel quale dovendo gestire anche le azioni future compaiono metodi che per ora sintetizzo solo.
Prima cosa vediamo esserci initTable e initWriteToFile, sono due routine usate per creare delle

strutture di supporto, la prima inizializza la symbolTable, mentre la seconda la scrittura sul file 'Strutture'.
Successivamente predisponiamo la lettura del file "Input.txt" che come già detto contiene il codice sorgente.
Infine chiamo il metodo yyparse che gestirà il parsing e si interfaccerà con l'analizzatore lessicale per farsi inviare i token man mano.
Se il parsing è andato a buon fine continuiamo con i passaggi che il programma dovrà compiere per arrivare alla fine del suo scopo, evalType gestirà l'analisi semantica, runCode l'interpretazione e infine chiudo la scrittura sul file.
Un'ultima aggiunta a quello che era lo standard: è stato aggiunto il campo intero linen al nodo come visto nella sezione precedente, è qui che lo riempiremo durante la creazione (metodo newnode) con la linea attuale del lexer che ricordiamo avere a disposizione tramite dichiarazione extern, questo ci permetterà anche nella semantica di indicare a quale riga viene trovato un certo errore.

ANALIZZATORE SEMANTICO

FILE COINVOLTI

Semantica.c, semantica.h, table.c, table.h, writeToFile.c, writeToFile.h

STRUTTURE COINVOLTE

Compare ovviamente l'albero astratto che è il nostro punto di partenza per poi fare tutta l'analisi, la sua definizione è un insieme di Pnode collegati tra loro tramite puntatori, come visto precedentemente.
Entry che è una struttura definita in table.h e corrisponde ad una "casella" della symbol table, i campi sono quelli definiti nelle slides del progetto di cui ne descrivo solo alcuni.
'ambiente' è un puntatore ad una Table ovvero ad un'altra symbol table/hash table, e corrisponderà allo scope locale di una funzione.
'dformali' è un array di puntatori ad altre Entry, queste saranno definite su una Table diversa da quella di partenza e corrisponderanno ai parametri della funzione.
'next' puntatore a nuova Entry, serve per costruire la linked list (spiego sotto).

SYMBOL TABLE

Spieghiamo in questo blocco come è stata implementata la symbol table.

Il codice è scritto nel file table.c, la funzione di hashing cuore di tutto il funzionamento è uguale a quella definita nelle slides, apro qui una parentesi per riportare l'unico problema sicuro che ho riscontrato ovvero che se la lettera iniziale è un carattere tipo 'è' il valore di ritorno è negativo.
Poi abbiamo due metodi insert e insertInto, la differenza tra i due è che uno ha una symbol table come parametro (nel codice Table è la symbol table) nell'altro invece no (viene usata quella globale), voleva essere una specie di overload di metodi.
Comunque una volta che il metodo viene chiamato computa la funzione di hashing sulla stringa

che corrisponde al nome della variabile o funzione, calcola l'object identifier tramite una variabile della table aumentata di uno e inizia il tentativo di inserimento.

Per creare la hashtable ho usato un array di linked list, il primo tentativo è di inserire la entry nella posizione dell'array corrispondente al valore di hash, questa però potrebbe già piena, nel qual caso inizia lo scorrimento della linked list.
Lo scorrimento può avere due possibili risultati, incorro in una entry che ha come chiave lo stesso valore che vorrei inserire, ovvero una variabile con quel nome già esiste, oppure trovo una posizione NULL che posso riempire.
La funzione restituisce un intero che corrisponde alla riuscita o meno dell'operazione, l'eventuale errore verrà generato dal chiamante.

Il metodo lookUp inizia a cercare la variabile nella tabella che gli viene passata come argomento, nel caso non venga trovata passa alla tabella globale. Questo meccanismo permette di rispettare lo shadowing delle variabili globali tramite quelle locali.
Nel caso non venga trovata alla fine viene restituito NULL e sarà compito del chiamante lanciare un eventuale errore.

Il metodo getOid è nato perché non era previsto che il metodo lookUp potesse avere un flag per controllare solo la tabella che gli viene passata e non anche quella globale, per non modificare chiamate vecchie ho scritto questo, il funzionamento è identico ma si ferma se non trova nella prima table.

DESCRIZIONE

Facendo un cenno di teoria la nostra grammatica di attributi è di tipo attributi memorizzati in strutture esterne, in particolare nelle Symbol tables viste prima.

Vediamo il tipico flow che percorrerà l'analizzatore:

Se ci sono variabili dichiarate viene chiamato il metodo varDeclList, questo inserisce nella Symbol table tutti gli identificatori che trova con il corretto tipo scritto nel codice sorgente. Se un ID esiste già viene chiamata la funzione di errore e il programma termina con il messaggio di errore corrispondente.

Se ci sono funzioni dichiarate viene chiamato il metodo funDeclList.
Ora ci saranno due tabelle che dobbiamo riempire, partendo da quella locale questa verrà creata e riempita con i parametri della funzione, in quella globale dobbiamo aggiungere l'esistenza di questa funzione collegare ad essa l'ambiente corrispondente (tabella locale) e far puntare l'array di dformali alle corrispondenti entry di quest'ultima.
A questo punto le Symbol tables sono complete, va controllato il corpo.
Per farlo alla fine del metodo chiamiamo body() che definiamo dopo e che controllerà la correttezza della stat-list della funzione

Come per le funzioni anche per il corpo principale verrà invocato il metodo body e passato il puntatore di inizio della stat-list del main.

Il metodo body è solo uno smistatore (implementato tramite switch) che reindirizza ogni stat del corpo alla routine di controllo di competenza.

I controlli sono quelli specificati nelle slides sulla semantica del progetto, aggiungo solo considerazioni non facilmente intuibili :
Ogni metodo riceve in ingresso un puntatore a table, questo perché potrebbe essere invocato per il controllo di una stat del main o di una funzione e gli è necessario sia poter accedere ai tipi delle variabili locali sia verificare la loro esistenza.

For-stat e while-stat modificano una variabile dentroCiclo, questa viene usata dal metodo breakStat per controllare se la definizione di una BREAK è avvenuta all'interno di un ciclo.

Una particolarità che si può notare è che ci sono due categorie di funzioni in questa fase, quelle che gestiscono tutto ciò che sta "sopra" il nodo expr e tutte quelle che gestiscono ciò che sta sotto.
A partire da expr in giù infatti ciò che ci aspettiamo dalla funzione è il tipo risultante, oltre al controllo che questo tipo sia ammissibile.
Facciamo un esempio, a = c + 6 * integer(2.5);
In questo caso abbiamo una stat, questa passando nel body viene riconosciuta essere un assegnamento (tramite NASSIGN_STAT) e viene chiamato il metodo predisposto, questo si occupa di valutare se l'assegnamento è legale, e il come è che chiama expr passandogli tutto quello che c'è a destra e aspetta un tipo.
La funzione expr ottenuto il puntatore da cui partire capisce, in base a cosa punta, se lui deve fare qualcosa o passare sotto e ritornare il risultato ottenuto dalla chiamata, questo procedimento si ripete finché arriviamo a factor, a questo punto potremmo ritornare un tipo oppure dover richiamare uno dei vari metodi soprastanti e la procedura si ripete.
Ciò che volevo indicare era come questi metodi fossero connessi tra loro e distinti dal secondo gruppo ('sovrastante') indicato prima.

ESECUZIONE

FILE COINVOLTI

Interprete.c, interprete.h, runStructure.c, runStructure.h, stringHeap.c, stringHeap.h, stringPoolStructure.c, stringPoolStructure.h

STRUTTURE COINVOLTE

STACK

Lo stack molto similmente a quanto consigliato sulle slides è composto dallo stack in sé e da una pila che tiene traccia dei record di attivazione chiamate ostack e astack.

L'ostack è un array di dimensione fissa di Ostackrecord, questa struttura contiene un campo HashType che indica il tipo e Value che tiene il valore.
Quando inizializziamo lo stack creiamo anche una variabile op che punta all'inizio dell'array, e continuerà a puntare, anche man mano che viene aggiunta, roba alla prima casella 'libera'.

L'astack è anch'esso un array di dimensione fissa, ogni sua casella è una struttura di tipo Astack con tre campi : nObjs contiene il numero di oggetti sullo stack appartenenti a quel record,

*startPoint che punta alla prima casella di Ostack appartenente al record, *table che punta alla symbol table di quella funzione.
Quando inizializziamo Astack creiamo anche una variabile ap che punta alla prima casella libera dell'array.

HEAP

L'unico tipo di dimensione variabile che Simpla contempla sono le stringhe, quindi l'heap in realtà contiene solo quelle.
Inoltre seppur la rimozione di stringhe non utilizzate sarebbe stata possibile non ho trovato un modo per ricompattare le informazioni senza che i puntatori si "rompessero" e quindi sono rimasto che quando qualcosa entra nello heap rimane per tutta l'esecuzione.

Per rendere possibile che due stringhe uguali puntassero alla stessa cosa in memoria (richiesto nelle specifiche del linguaggio) è stato implementato un meccanismo che ad alto livello è simile allo string pool di Java (molto semplificato).

Il primo problema è tenere traccia di tutte le stringhe che ci sono già in memoria, questo è fatto da una hashtable in struttura uguale a quella utilizzata nella symbol table.
Nel file stringPoolStructure abbiamo le funzioni identiche a quelle già viste, cambia però l'inserimento perché non abbiamo complicazioni come quelle avute nel dover inserire una funzione con i suoi parametri e uno scope locale.
Differisce anche il comportamento quando trova un elemento con la stessa chiave, infatti ritorna il puntatore alla stringa nell'heap già esistente con gli stessi caratteri.
Per quanto riguarda le entry di questa hashtable ogni elemento avrà un puntatore a stringa (che punterà all'heap) e un puntatore ad un eventuale elemento successivo (per creare la linked list).

Il secondo problema è avere uno spazio di memoria per memorizzare effettivamente le stringhe a cui poi vogliamo puntare.
Nel file stringHeap abbiamo un array di char di dimensione fissa, e un puntatore alla prima casella libera di questo array.
L'unico metodo utile di questo file prende in input un puntatore a stringa, copia carattere per carattere spostando man mano il puntatore interno alla prima casella libera e termina ritornando un puntatore all'inizio della parola appena aggiunta.

DESCRIZIONE

Alcuni dei file creati per questa fase potevano essere evitati nel senso che il codice contenuto poteva essere incluso in file di sezioni passate, ho voluto però tenere, in particolare questa fase, il più indipendente possibile dal resto sia per chiarezza sia per permettere senza troppi sforzi di staccarla e rendere in teoria possibile la divisione in compilatore e interprete.

VARDECLLISTEX

Questo metodo viene chiamato ogni volta che viene creato un nuovo record di attivazione (compreso quello del main) e si occupa di aggiungere sullo stack le variabili dichiarate di quel particolare corpo, inoltre per ragioni che si vedono dopo l'ordine con cui vengono aggiunte è uguale al loro object id.

Ad esempio a,b,c : integer con valori Oid nella symbol table 1,2,3 verranno aggiunte nello stack proprio in questo ordine.

Aggiungo qui che quando un record viene aggiunto ad ostack viene chiamata una funzione che oltre ad aumentare il puntatore aumenta la variabile nObject relativa al record di attivazione attivo in quel momento.
Stessa cosa quando si diminuisce ma al contrario.

Prima di spiegare tutta la parte legata al body esploro prima le funzioni da expr in giù in quanto sono quelle più particolari e senza le cui spiegazioni è difficile descrivere il resto.

EXPREX E 'FIGLI'

Il funzionamento dei metodi bassi diciamo è molto simile a quello che era nella sezione di semantica, la differenza è ciò che producono mentre rimane uguale il flow.

In pratica quando viene chiamata expr e gli viene passato il puntatore dal quale iniziare a lavorare possono succedere due cose : riconosce un tipo 'AND' o 'OR' e deve lavorarci oppure passa sotto e lui ha finito (sono void in questo caso).

Chi e quanti debbano lavorare per produrre il risultato è variabile ma non troppo importante, ciò che conta è il meccanismo con il quale passano ciò che hanno ricavato ai metodi che li hanno chiamati e questo avviene tramite lo stack.

Il comportamento aspettato è che una volta chiamato un metodo basso quando questo terminerà e restituirà il controllo al metodo chiamante sullo stack ci sia il/i valori prodotti. A questo punto si elaborano questi dati e si memorizzano o si stampano a video o ciò che vuole quella particolare stat.

Ad esempio se una stat è 'a = 5 + 3 * a;' esploriamo il percorso del programma.
Ipotizzando che l'unica altra stat sia a = 5; definita precedentemente allora sullo stack avremo che inizialmente c'è solo un valore [a|INTEGER] ('a' è variabile globale, viene aggiunta prima di eseguire il body) e sull' astack un solo record con nObj = 1 (il main).
La funzione bodyex capisce che è una stat di tipo assign e chiama la funzione designata, questa dovrà fare due cose : computare il RHS e memorizzarlo nella variabile 'a' (i tipi sono sicuramente compatibili perché il type checking si verifica nella semantica).
Ciò che succede è che viene chiamata exprex passandogli il puntatore alla RHS, questa particolare espressione non contiene elementi che matchano in expr quindi sarà passata sotto man mano finché non trova il metodo che può risolverla (anche chiamando altri metodi a sua volta).
Quando viene restituito il controllo ad assignStatex lei sa che sullo stack in questo momento c'è il risultato della parte destra e le rimane solo da assegnarlo alla LHS (il funzionamento è scritto nella parte successiva).

RECUPERO ID

Una piccola precisazione prima di esplorare gli altri metodi, il funzionamento di recupero di una variabile è uguale in tutta la sezione e lo esploro qui. (sia per modificarne il valore che per leggerlo).

Ho spiegato prima come fosse importante e rispettato che l'aggiunta allo stack delle variabili dichiarate avvenisse con lo stesso ordine che i loro Oid avevano nella symbol table, e il motivo lo troviamo qua.
Quando voglio recuperare una variabile chiedo alla symbol table l'oid della stessa, chiedo all'attuale record di attivazione il suo punto di partenza sullo stack, aggiungo ad esso l'oid e arrivo nella casella contenente la variabile voluta.
Tra l'altro questo è l'unica dipendenza pesante che la parte di interpretazione e il resto hanno tra loro, avrei voluto e potuto evitarla ma mi pareva eccessivo copiare anche queste informazioni creandone un doppione, è comunque possibile farlo anche ora senza troppa fatica quindi credo rimanga il concetto di flessibilità.

Il meccanismo così com'è funziona solo se la variabile è locale, per permettere anche l'utilizzo di identificatori globali basta aggiungere che se la symbol table con la sola tabella che gli abbiamo passato (la locale) non trova l'identificatore il record di attivazione a cui dobbiamo chiedere il punto di partenza sullo stack non è quello attivo (l'ultimo sulla pila) ma il primo in assoluto (il main) e da li il procedimento è il medesimo.
Qui si spiega la presenza di getOid che come abbiamo spiegato prima sembra a prima vista un metodo doppione del file table.c

BODYEX
Similmente alla sua controparte nella semantica anche questo metodo è solo uno smistatore che con uno switch chiama la procedura corretta in base al tipo di stat.

L'if che ha come condizione funInterrupt verrà spiegato nella funzione funcCallex.
L'if con toExit in breakStatex.

WHILESTATEX
Fa risolvere la condizione di if ad exprex, legge il primo elemento della pila se è vero esegue il corpo tramite bodyex sennò esce, in entrambi i casi rimuove un elemento dallo stack che era la condizione booleana aggiunta precedentemente.

BREAKSTATEX
Pone un flag (toExit) ad 1 ed interrompe l'esecuzione del body, in for e while stat finita l'esecuzione del corpo e prima di iniziare un nuovo giro si controlla se l'uscita è stata forzata da questa istruzione e, in caso positivo, viene terminata l'esecuzione.

ASSIGNSTATEX
Risolve il ramo destro tramite exprex e chiama una routine.
In pratica con la prima istruzione poniamo in cima allo stack la RHS dell'assegnamento e la routine che spieghiamo dopo non fa altro che assegnare al nome che gli passiamo il valore più in alto sullo stack.

WRITESTATEX
Scorre tutte le espressioni che gli sono passate come argomento, ognuna viene risolta con l'apposita funzione che mette il valore in pila, nello stesso ciclo prendiamo il primo valore e in base al tipo lo stampiamo sullo stdout.

Come con tutte le altre funzioni quando un elemento sullo stack aggiunto tramite una sua chiamata o una sua istruzione non è più utile viene rimosso.

READSTATEX

Scorro tutte le variabili che si vogliono riempire, per ognuna tramite la symbol table ne ricavo il tipo, controllo che l'input sia legale (sezione funzioni di supporto) per il tipo della variabile e lo converto da stringa a quello che serve.

Nel case String vediamo che il procedimento è leggermente strano, questo perché devo aggiungere la stringa alla string pool e puntare all'heap non più alla locazione fornita da c, questo avviene tramite il metodo addString che ho spiegato nella sezione Strutture coinvolte.

FORSTATEX

Faccio solo una volta l'assegnamento definito nel for (es. i=0), poi entro in un ciclo.
Per prima cosa nel ciclo controllo che la condizione sia ancora vera, se lo è eseguo il corpo del for sempre tramite bodyex, infine devo aumentare la variabile di controllo di 1, nel farlo chiamo una routine getValueVarStack che spiegherò dopo ma si può riassumere dicendo che restituisce il valore della variabile passata come argomento.

RETURNSTATEX

Pongo a 1 il flag funInterrupt che ci servirà dopo per sapere che devo smettere di eseguire il corpo di una certa funzione.

Dopodiché se il return restituisce qualcosa computo quell'espressione, aggiungo quel valore ad una specifica posizione dello stack (spiego dopo) e rimuovo da astack il record di attivazione attivo in questo momento con il conseguente cambiamento anche del puntatore op che tiene traccia della prima casella libera su ostack.
Per ricavare la nuova posizione di op semplicemente partiamo dalla prima posizione del record di attivazione precedente e gli aggiungiamo il numero di oggetti che gli appartengono.

FUNCALLEX

Se la funzione chiamata non è void creo uno spazio sullo stack che accoglierà il suo valore di ritorno (questa casella sarà ancora parte del record di attivazione chiamante).

Creo il nuovo record di attivazione e gli do la symbol table locale della funzione, non lo aggiungo subito ad astack ma segno comunque la posizione attuale di ostack come quella di partenza(la posizione a cui punta ora ostack è quella che verrà segnata come di partenza per questo nuovo record che stiamo creando).E quindi non muovo il puntatore ap.

Aggiungo allo stack i parametri e man mano che li inserisco computo il valore passato come argomento, ed è qui il motivo per cui il nuovo record (astackrecord) non è stato subito aggiunto, serviva che exprex potesse risolvere il valore dell'argomento avendo la visione del record precedente.

A questo punto possiamo aggiungere il nuovo record e aggiorniamo il valore di nObj con le entry aggiunte appena sopra.

Aggiungo allo stack la variabili locali dichiarate nella funzione.

Infine eseguo il corpo della funzione con bodyex, nello stack c'è già tutto quello che gli serve.

INTERRUZIONE ESECUZIONE FUNZIONE

Al normale funzionamento si aggiunge il caso in cui ci sia un'istruzione di return prima dell'ultima riga del corpo.

Come abbiamo visto prima quando incontriamo un return viene posto un flag ad 1, similmente a quanto accade con il flag toExit, finché è ad 1 il body continuerà ad eseguire il return salendo la pila di chiamate e tornando rispettivamente in funcCall o in while/for stat, è qui che dovremo disattivare il flag e compiere le azioni che servono.
Nel caso della funzione ciò che ci serve sapere è che se il flag è ad 1 vuol dire che il valore di ritorno è già stato copiato nello stack nell'ultima posizione del RA precedente e il puntatore op aggiornato, mentre nel caso contrario (avviene solo in caso di funzione void senza return esplicito) vuol dire che dovremo fare noi sia la rimozione dell'RA che l'aggiornamento del puntatore di ostack (di solito se ne occupa returnStatex).

FUNZIONI DI SUPPORTO

GETVALUEVARSTACK

Cerco l'Oid della variabile dalla symbol table (legata al RA attualmente attivo), se non lo trovo qua uso la symbol table globale che ottengo tramite la funzione getGlobale().

Dato l'Oid se lo sommo alla posizione di partenza sull'ostack del RA attuale ottengo la casella contenente il valore e il tipo della variabile cercata.

ISSOMETHING

isInt, isReal, isBool sono semplici metodi di supporto che mi permettono di lanciare un errore nel caso in cui l'input della read non sia dello stesso tipo della variabile che deve essere riempita.

CAMBIAVALINSTACK

Otteniamo la posizione nello stack dove risiede la variabile che vogliamo modificare nello stesso modo di getValueVarStack e ad essa associamo il valore che leggiamo in cima alla pila.

Prima di terminare puliamo anche la cima dello stack perché non è più utile.

AUMENTA/DIMINUISCIOP

Sposto il puntatore op e in base al tipo di chiamata aumento o diminuisco la variabile nObj del RA attuale.