Laboratorio di Compilatori

Simone Federici Zafiropoulos Vassilios

17 maggio 2004

Indice

1 Introduzione

Obbiettivo Costruire in java un compilatore per il linguaggio Tiger.

Premessa Implementare un compilatore è una cosa ardua e complessa, A.W. Appel con il libro Modern Compiler Implementation in Java, ci ha semplificato non poco la vita. Un ringraziamento al Prof. Paolo Bottoni che sempre con grande disponibilità ci ha seguito in questo progetto.

Questa relazione è composta da appunti, schemi e commenti rivisitati e messi in ordine a fine progetto. Inizieremo schematizzando i passi importanti per la realizzazione di un *Compilatore*, ossia l'implementazione di un programma in grado di *tradurre* sentenze scritte in un linguaggio di programmazione in codice eseguibile equivalente. In particolare l'implementazione di un compilatore per *Tiger*, un ottimo linguaggio didattico che è composto da tutti i costrutti di rilievo che troviamo solitamente nei linguaggi più comuni. Successivamente andremo sempre più nello specifico analizzando le scelte focali che abbiamo effettuato. Il compilatore è stato implementato in `Java` con l'ausilio di due programmi, **JLex** e **CUP**. Il progetto è stato portato avanti sia in ambiente linux che windows grazie alla portabilità di Java.

Compilazione Il processo di compilazione deve rispettare un vincolo molto importante, produrre codice eseguibile equivalente al programma originario. In semantica denotazionale, diremmo che la funzione denotata del programma originario e quella della sua traduzione in codice eseguibile devono avere lo stesso significato, ossia con lo stesso input devono dare lo stesso output.

Per garantire tale obiettivo, il processo di compilazione viene suddiviso in diverse fasi, dove in ogni fase si garantisce l'equivalenza tra la semantica del codice di partenza e quella del codice di arrivo. La prima fase prende in input il programma scritto in Tiger e restituisce un codice intermedio. L'ultima fase restituisce il codice eseguibile finale.

Analisi di un Linguaggio Un linguaggio è un insieme di sentenze che sono corrette da un punto di vista **sintattico** (generabili e analizzabili da regole formalizzate) e **semantico** (hanno significato). Un linguaggio è composto quindi da un **Lessico** (vocabolario), da una **sintassi** e da una **semantica**.

Un linguaggio di Programmazione ha queste tre componenti, in più si sa che ogni sentenza prescrive la computazione di determinate funzioni, quindi è non ambiguo. Il Compilatore è precisamente lo strumento attraverso cui si può stabilire se una sentenza di un linguaggio di programmazione è corretta o meno e, se è corretta, quale è il suo significato in termini di computazione, ossia quale è la sua semantica.

Analisi Lessicale **JLex** è una utility scritta in java realizzata appositamente per costruire in pochi passi un analizzatore lessicale (DFA). Scriverne uno a mano sarebbe stato molto lungo e noioso. **JLex** prende in input delle specifiche e restituisce un programma java in grado di trasformare uno stream di caratteri in una sequenza di token.

Bisogna notare però che gli identificatori, le costanti e i valori sono spesso infiniti. Ossia non posso elencare tutte le possibili combinazioni di lettere, nume-

ri, underscore e altri caratteri alfanumerici come possibili parole del linguaggio. Per questo vengono utilizzate le espressioni regolari, attraverso l'opportuna definizione di linguaggi regolari. Le forme corrette di questi linguaggi possono quindi essere riconosciute da automi a stati finiti.

Analisi Sintattica Un linguaggio di programmazione è solitamente definito da una struttura sintattica che può essere espressa nella forma di una grammatica libera dal contesto. Noi siamo interessati a definire la struttura sintattica di tutti i possibili programmi *Tiger* corretti e quindi a saper costruire tutti i possibili alberi da essi derivabili. In poche parole siamo interessati a definire la sintassi concreta di *tiger* mediante una apposita grammatica libera dal contesto. Per far ciò abbiamo bisogno di un analizzatore sintattico che analizzi la correttezza sintattica del programma in input.

L'analizzatore sintattico prenderà in input la sequenza di token emessa dall'analizzatore lessicale e un token alla volta costruirà uno e un solo albero della sintassi astratta del programma Tiger in input. Per costruire l'analizzatore sintattico abbiamo utilizzato **CUP**, una utility messa a disposizione a questo scopo. **CUP** prende in input un insieme di specifiche e restituisce un analizzatore sintattico. È importante pensare che l'analizzatore sintattico prodotto genererà solo alberi non ambigui.

Analisi Semantica Ora che abbiamo un albero della sintassi astratta è tutto più facile, possiamo scorrere avanti e indietro l'albero per controllare che il significato del programma sia corretto. Per esempio una variabile non può essere utilizzata se non è stata precedentemente dichiarata, non posso fare operazioni numeriche con delle variabili di tipo String, etc.

Codice Intermedio Abbiamo tradotto l'albero di sitassi astratta in un albero di rappresentazione intermedia, allo scopo di rendere più modulare il compilatore. Infatti a partire dall'albero di rappresentazione intermedia si può ricavare codice macchina per qualsiasi architettura. Contemporaneamente creiamo le strutture dati di supporto per le funzioni (`Frame`). L'albero cosi creato si riferisce ad una macchina astratta con numero infinito di registri.

Ottimizzazione Il codice intermedio viene trasformato in un insieme di blocchi di pseudo-istruzioni. Questa parte è stata fornita da Appel.

Generazione del Codice Dall'albero del codice intermedio ottimizzato ricaviamo un programma in assembler che usa l'insieme di istruzioni fornito dall'emulatore **SPIM**. Il programma ottenuto si riferisce comunque ad una macchina con un numero infinito di registri. Il compito di rendere correto il programma, eseguibile cioè dall'emulatore, è lasciato al modulo RegAlloc. La generazione del codice avviene cercando nell'albero del codice intermedio dei *pattern*, ossia delle porzioni di albero ben definite. Per ognuno di questi viene generata una o più istruzioni. Questa pratica si chiama *tassellamento* dell'albero. Esistono due tipi di tassellamento. Ottimo e ottimale. Il tassellmento ottimo produce l'insieme di instruzioni di costo più basso possibile. Si basa su un modello di costo, che però non è esatto, in quanto istruzioni vicine possono interferire. Il tassellamento

ottimale cerca di avvicinarsi all'insieme ottimo. Per costo si intende o la somma dei costi singoli in termini di tempo, oppure il numero di istruzioni complessivo.

Allocazione dei Registri L'allocatore dei registri usando le informazioni del modulo di analisi di vita genera il codice finale. Lo fa costruendo un grafo che denota quali registri possono essere usati per quali variabli(registri della macchina astratta) e cercando di colorarlo con un numero di colori pari a quello dei registri reali.

2 Dall'analisi lessicale al codice intermedio

Bisogna tenere bene a mente che cos'è un compilatore, come gia detto è scritto in java, quindi è possibile usarlo su qualsiasi piattaforma. Si da in pasto un file dove è stato scritto un programma in linguaggio *Tiger*, il compilatore lo controlla lessicalmente, sintatticamente e semanticamente e se è tutto in regola produce codice assembler per macchine **MIPS**. Questo codice è eseguibile su macchine **MIPS** e noi lo abbiamo sperimentato sull'emulatore **SPIM**.

2.1 Analisi Lessicale

Come già introdotto precedentemente l'analisi lessicale è molto semplificata usando **JLex**. Dando in pasto a **JLex** Tiger.lex, ossia le specifiche del Tiger, viene restituito l'analizatore lessicale `Yylex.java`. In appendice si trova il codice della specifica.

Comment Per gestire i commenti del programma sorgente *Tiger* abbiamo introdotto uno stato COMMENT. Il parser inizialmente si trova nello stato YYINITIAL. Appena troviamo la sequenza di caratteri `/*` il parser entra nello stato COMMENT e ci resta finche non vengono chiusi i commenti. In questo stato tutti i caratteri letti vengono scartati fino alla sequenza `*/`.

Fin qui tutto facile, ma come gestire i commenti annidiati? Abbiamo introdotto una variabile di stato comment_count=0 che viene incrementata ogni volta che si aprono i commenti e viene decrementata ogni volta che si chiudono. Il ritorno allo stato YYINITIAL viene quindi vincolato da questa variabile. Solamente se ritorna ad avere valore zero verrà effettuato il cambio di stato.

2.2 Analisi sintattica - Parser

Il processo di parsing è molto complicato. Concerne l'implementazione da zero di un algoritmo Look-Ahead-LR, cosa molto tediosa. Il **CUP** è lo strumento java che prendento in input delle specifiche, produce un parser *LALR*. Le specifiche le possiamo dividere in tre parti, codice java aggiuntivo, grammatica libera dal contesto e precedenze. Inizialmente abbiamo scritto la grammatica del linguaggio Tiger ma alla prima stesura **CUP** ci ha fatto notare molti conflitti Shift/Reduce e Reduce/Reduce. Per risolvere questi problemi abbiamo usato un forte strumento di **CUP**, le precedenze.

Blocchi di Dichiarazioni La definizione di tipi mutualmente ricorsivi in *Tiger* è stata la chiave di una scelta importante che ha dato una svolta significativa al progetto. I tipi mutualmente ricorsivi sono dichiarati in una sequenza consecutiva di tipi senza interruzioni da dichiarazioni di variabili o di funzioni. Questa definizione ci lasciò perplessi. Perche Appel dava una limitazione cos grande? Forse perchè era molto complicato gestire in un modo più generico queste definizioni di tipo (questa è la rispista secondo noi).

Noi abbiamo scelto di non tenere conto di questa limitazione. Anche se le dichiarazioni mutualmente ricorsive sono interrotte da altre dichiarazioni il comilatore le acccetta senza segnalare l'errore. L'unica, ovvia, limitazione è che siano nello stesso blocco dichiarativo. Lo stesso discorso vale per la definizione di funzioni mutualmente ricorsive. Arrivati al codice intermedio e riflettendo

su questa scelta, siamo arrivati alla conclusione che la difficolta incontrata non è stata eccessivamente elevata. Alcune parti (la scrittura della grammatica) si sono semplificate ed altre (l'analisi semantica) si sono complicate notevolmente.

Operazioni Matematiche Conosciamo bene l'ambiguità in una grammatica con le quatro operazioni matematiche se non adottiamo una soluzione particolare. Abbiamo scelto di non complicare la grammatica perché aggiungendo queste due righe tra le specifiche di **CUP** l'ambiguità viene risolta.

```
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
```

In questo modo $3-2-5$ verrà tradotto in un unico albero di sintassi astratta **-(-(3, 2), 5)**. L'ordine delle direttive tradurrà $3+4*5$ sia **+(3, +(4, 5))**. I token delle direttive più in basso hanno precendenza maggiore.

Uminus Non esiste un token chiamato UMINUS, cioè l'analizzatore lessicale riconosce il carattere − con il token MINUS. A volte però l'operatore di sottrazione deve essere considerato come un operatore unitario e quindi avere una precedenza totale su qualsiasi altra operazione, il che vuol dire che bisogna dargli una precedenza più forte. **CUP** ci da la possibilità di dare questa precedenza "speciale" aggiungendo un nuovo non terminale UMINUS e usando questa sintassi.

```
precedence left UMINUS;
Exp ::= MINUS exp  %prec UMINUS
```

La prima riga è il tipo di precedenza che ha il token UMINUS (è inserita per ultima tra la lista delle precedenze in quanto è la più forte). La seconda direttiva a destra della produzione (%prec UMINUS) dice al parser che in questo caso la precedenza non è quella solita del MINUS ma quella dell'UMINUS.

Dangling Else Il token ELSE da un conflitto Shift/Reduce riguardo a quale "if-then" è legato. Ad esempio:
`if E1 then if E2 then E3 else E4`
a quale "if-then" è riferito l'else? Il parser, se non gli diamo nessuna direttiva, riduce su E3 e si trova in uno stato dove può fare una *reduce* `[if E1 then E2.]` o fare uno *shift* su `[if E1 then E2 else. E3]`. **CUP** ignorerebbe questo conflitto e favorirebbe lo *shift* anziche il *reduce* se usassimo `expect 1`. Per un fatto di pulizia, ma senza voler complicare ulteriormente la grammatica, abbiamo scelto di inserire questa precedenza

```
precedence nonassoc ELSE;
```

in modo da dare precedenza all'ELSE su gli altri terminali.Per questo motivo il parser effettuerà uno *shift* anziche un *reduce*. Ci siamo chiesti se usare nonassoc left o right ma abbiamo concluso che nel caso dell'else, l'uno vale l'altra, infatti in tutti e tre i casi si ottiene che lo shift è preferito al reduce ossia che l'else viene attribuito all'if più interno.

Altre precedenze

```
precedence nonassoc ASSIGN;
precedence left OR;
precedence left AND;
precedence nonassoc EQ, NEQ, GT, LT, GE, LE;
```

L'assegnazione ha una precedenza non associativa, ossia non posso fare `A := B := C`. Abbiamo dato precedenza all'AND sull OR per convenzione, e associatività a sinistra (questo serve solo a levare l'ambiguità, anche dando associatività a destra il risultato dell'espressione non cambierebbe) `A OR B AND C AND D` viene tradotto come `A OR ((B AND C) AND D)`. Le operazioni di confronto $=, != , >, <, >=, <=$ non hanno associatività tra loro, solo una di esse può essere usata in una espressione a meno che non ci sia un altro terminale in mezzo. `3 = 4 > 5` è sintatticamente errato. Mentre è lecito `3 = 4 AND 5 > 6`.

Break L'istruzione BREAK è stata curiosa. In un primo momento il semplice token BREAK lo consideravamo una istruzione corretta. Poi ci siamo accorti che pur essendo una delle parole chiave del linguaggio *Tiger*, si doveva trovare solamente all'interno di cicli WHILE o FOR. Siamo arrivati ad un bivio, potevamo duplicare l'intera grammatica aggiungendo il non terminale `expC` che veniva chiamato da un WHILE o da un FOR e mettere il BREAK solo come figlio di `bodyC` oppure considerare questo errore come un errore semantico, ignorandolo in questa fase per poi risolverlo successivamente nell'analisi semantica.

Infine abbiamo pensato che si tratta di un errore sintattico e che era giusto risolverlo duplicando la grammatica.

Error Recovery Questa fase è stata massacrante infatti mentre studiavamo cos'era l'error recovery e come si usava in **CUP** testavamo quello che imparavamo su dei semplici programmi. Catastrofe perché avevamo una versione errata di **CUP**. In un primo momento, quindi, abbiamo saltato questa fase e siamo andati a costruire l'albero nel capitolo successivo. Finito il capitolo quattro siamo tornati sull'error recovery e c'è giunta la grandiosa notizia che stavamo usando una versione di **CUP** errata. Abbiamo cominciato inserendo il token speciale error nelle produzioni più semplici:

```
exp ::= exp PLUS error
```

Da questa produzione abbiamo capito immediatamente che era importantissima la precedenza di questo token. Abbiamo concentrato le idee sul cercare di andare avanti con il parser appena trovato un errore, per questo abbiamo scelto di cercare fare *shift* sul token error prima possibile. Abbiamo quindi dato la precedenza assoluta al token error (è l'ultimo della lista). In questa fase abbiamo fatto molti esempi con delle istruzioni sbagliate e vedevamo dove e come il parser reagiva e come recuperava l'errore. Prima siamo partiti dalle semplici istruzioni fino a poi fare dei veri e propri programmi.

Abbiamo seguito sempre la stessa tecnica per tutte le produzioni, per esempio la produzione del FOR:

```
FOR:f ID:i ASSIGN expC:e1 TO expC:e2 DO error
FOR:f ID:i ASSIGN exp:e1 TO error
FOR:f ID:i ASSIGN exp:e1 error
FOR:f ID:i ASSIGN error
```

```
FOR:f ID:i error
FOR:f error
```

Come possiamo vedere da questa scomposisizione della produzione FOR abbiamo considerato tutte le possibilità. Nel momento esatto in cui viene trovato il token error vengono scartati tutti i token successivi finché non ne viene trovato uno che appartiene all'insieme FOLLOW. Per visuaizzare un albero della sintassi astratta completo, ogni qual volta incontriamo un errore in una produzione, associamo una foglia ben precisa di errore. Per l'esempio precedente è `new ErrorExp(fleft, FOR ID := error)`.

Questo perche abbiamo scelto di controllare anche gli errori semantici anche se troviamo un errore sintattico. Questa scelta è un po' azzardata in quanto un errore sintattico potrebbe avere grosse conseguenze su tutto il resto del programma, È però vero anche che alcuni errori sintattici non hanno grosse conseguenze. Per questo abbiamo scelto di completare un albero anche se con foglie `Error`. Vediamo alcuni esempi
Programma n.1
Errore 1: tipo di ritorno della funzione f non coincide con quello dichiarato
Errore 2: entrambi gli argomenti della diseguaglianza sono void
Errore 3: tipo del ramo else non coincide con il ramo then
Errore 4: end mancante alla fine del file.

```
let
  var x:=3
  type boolean = {true:string, false:string}
  var z:boolean := boolean{true="uno", false="zero"}
  function f(x:int) =
    let
      var y:=0
    in
      y
    end
in
  for i:=10 to "\" do
    (
      i := i - 1;
      if(f(4)>f(10)) then 5 else z.false
    )
```

```
1) ERRORE: file:tiger/testcases/errori.tig::29.6--> Parentesi o
blocchi non chiusi alla fine del file
    )
      ^
2) ERRORE: file:tiger/testcases/errori.tig::19.3--> Il tipo
aspettato (Int) non coincide con quello restituito (Void)
  function f(x:int) = let
   ^
3) ERRORE: file:tiger/testcases/errori.tig::25.16--> E' Richiesto
un Intero
  for i:=10 to " " do
```

```
               ^
4) ERRORE: file:tiger/testcases/errori.tig::27.7--> La variabile
non puo' essere modificata (Assign)
      i := i - 1;
       ^
5) ERRORE: file:tiger/testcases/errori.tig::28.10--> E' Richiesto
un Intero
      if(f(4)>f(10)) then 5 else z.false
          ^
6) ERRORE: file:tiger/testcases/errori.tig::28.15--> E' Richiesto
un Intero
      if(f(4)>f(10)) then 5 else z.false
                ^
7) ERRORE: file:tiger/testcases/errori.tig::28.34--> I tipi sono
incompatibili (Then-Else)
      if(f(4)>f(10)) then 5 else z.false
                                    ^
```

In alcuni casi però l'error recovery non funziona correttamente. Vediamo:

```
let
  var x:=4
  2
end

ERRORE: file:errori.tig::3.2--> Errore di sintassi: #4 INT
 2
^
Couldn't repair and continue parse at character 591 of input
```

In questo esempio ci sono due errori. Il primo è un'espressione intera in un blocco dichiarativo ed il secondo è la parola chiave IN mancante. Il parser non riesce a recuperare tutti gli errori prima della fine del file al qual punto si arresta irrimediabilmente.

Pareggio blocchi e parentesi Abbiamo notato che a causa di parentesi o blocchi non chiusi l'error recovery falliva, specialemente negli ultimi token del programma. Per questo si è deciso di controllare e pareggiare automaticamente eventuali blocchi o parentesi non chiusi, durante l'analisi lessicale.

Abiamo introdotto una pila che ci da una informazione aggiuntiva sulla parentesi aspettata. Dopo avere cercato di raccogliere il maggior numero di errori, ci accorgevamo che se mancava una parentesi o l'END che chiudeva il LET, il parser per recuperare l'errore scartava tutto il blocco errato. Aggiungendo una parentesi o l'END per chiudere un blocco, e dando quindi al parser solo blocchi chiusi, l'errore era più circoscritto e spesso il parser andava più avanti recuperando se c'erano errori successivi.

Questa pila è stata realizzata passando un parametro in **JLex** (%function nextToken2) in modo che l'analizzatore lessicale dia il token successivo chiamando la funzione nextToken2(). Il **CUP** però chiama la funzione nextToken(),

abbiamo quindi aggiunto il codice della funzione nextToken() che si occupa dell'aggiunta di un eventuale Token atraverso una coda. Piu' precisamente ogni qual volta che l'analizzatore lessicale trova una parentesi aperta, inserisce in pila il token di parentesi chiusa. Ogni qual volta c'è una parentesi chiusa, prima di passare il token al parser, si controlla che in cima alla pila ci sia proprio quel tipo di parentesi chiusa. Se la pila è vuota, viene lanciato un errore (parentesi di troppo) e richiamato ricorsivamente nextToken(). Se invece non è vuota ma il token in cima è diverso dal token trovato, viene lanciato un errore e viene passato il token in cima alla pila mentre il token appena letto viene messo nella coda. La prossima volta che il parser chiederà un token, visto che la coda non è vuota, si prenderà il token presente nella coda.

```
Parsing
------- CUP v0.10k Parser Generation Summary -------
  0 errors and 0 warnings
  46 terminals, 26 non-terminals, and 233 productions declared,
  producing 427 unique parse states.
  0 terminals declared but not used.
  0 non-terminals declared but not used.
  0 productions never reduced.
  0 conflicts detected (0 expected).
  Code written to "Grm.java", and "sym.java".
----------------------------------------------------- (v0.10k)
```

Finalmente raggiunto un risultato più che accettabile, ossia una grammatica senza conflitti e senza termiali, non terminali e produzioni inutili (mai ridotte) ci siamo fermati più che soddisfatti del processo di parsing.

2.3 Sintassi Astratta

Come accennato in precedenza questa fase noi l'abbiamo eseguita prima di fare l'error recovery. Si tratta di costruire l'albero della sintassi astratta parallelamente al processo di parsing. Si tratta di aggiungere la porzione di albero della sintassi astratta corrispondente ad ogni produzione.

```
exp ::= INT:n  {: RESULT = new IntExp(nleft, n.intValue()); :}
```

Questa semplice produzione con le sue direttive, dice chiaramente che una espressione può essere un intero ossia una foglia dell'albero astratto. Il valore `nleft` è una valore che **CUP** prende dai token passati dall'analizzatore lessicale e corrisponde alla posizione del token nello streaming di input (serve per avere le cordinate precise nel caso si verificasse un errore). Senza voler levare nussun merito a nessuno dobbiamo però ammettere che questa fase è cosi spiegata e aiutata dal package `Absyn` fornito da Appel che la realizzazione è stata abbastanza banale. Unica eccezione, già spiegata in precedenza, è la creazione di tre foglie particolari: ErrorExp, ErrorDec e ErrorVar le quali sono dei chiari messaggi di errore che nello stampare l'albero mettono ben in risalto il blocco di errore. Da notare che gli unici terminali che vengono trasformati in una foglia immediatamente, sono INT e STRING infatti sono gli unici ad essere tipati. Tutti i non terminali invece hanno, come tipo, un nodo dell'albero astratto, ossia una classe del package `Absyn` fornito da Appel.

2.4 Analisi Semantica, Livelli, Frame e Codice Intermedio

Questa parte, a nostro avviso, è stata di forte impatto. Fino ad adesso ci eravamo confrontati con strumenti, come **JLex** e **CUP**, che configurati in un certo modo creavano per noi i programmi da usare. Non eravamo ancora entrati in contatto vero con java. Per questo diciamo che ha avuto un impatto forte, perchè ci siamo trovati a dover implementare Semant.java quasi da zero.

La fase di analisi semantica prende in ingresso dichiarazioni ed espressioni e controlla che siano ben tipate. Per questa ragione introduciamo due tipi di ambienti, quello dei tipi e quello dei valori. Nel secondo ci sono sia le variabili che le funzioni. A questo proposito ricordiamo che se in un ambiente viene definito un nome già presente, il nuovo prenderà il posto di quello vecchio. Ora introduciamo il concetto di scope: ogni qual volta si apre un nuovo blocco, un nuovo ambiente viene creato. Tutti i nomi dichiarati successivamente avranno validità solamente fino alla chiusura del blocco. Ogni volta che si usa il costrutto LET IN END si apre uno spazio di dichiarazioni che avranno validità solo all'interno del corpo di istruzioni tra IN ed END. È quindi necessario aprire un nuovo scope sia nell'ambiente dei tipi che in quello dei valori fino al termine dell'istruzione LET. I tipi base STRING e INT sono precaricati nell'ambiente dei tipi. Come ci ha consigliato Appel, in un primo momento non abbiamo considerato i tipi e le funzioni ricorsive. Da ricordare poi la nostra scelta dei blocchi di dichiarazione nella quale il nostro compilatore accetterà qualsiasi dichiarazione di tipo o funzione ricorsiva che sia contenuta tutta nello stesso blocco di dichiarazioni.

L'analisi semantica vera e prorpia viene eseguita dalla classe `Semant`. Al costruttore viene passato come parametro attuale l'albero di sintassi astratta ottenuto precedentemente. Quattro funzioni, `transExp, transVar, transDec` e
`transTy`, vengono chiamate ricorsivamente a partire dalla radice dell'albero. Il compito di `Semant` è di controllare i tipi delle espressioni dell'albero e di tradurle in sottoalberi del codice intermendio.

`transExp` legge le espressioni, `transDec` i blocchi di dichiarazioni etc. Ad esempio l'espressione `LetExp` del tipo "LET d IN b END" lancerà `transDec(d)` e una `transExp(b)` e via di seguito. Iniziando con `transExp(program)` si legge tutto l'albero. Mentre si attraversa l'albero come descritto sopra, si controlla la semantica, ossia che tutto sia ben tipato.

LetExp Il modo per dichiarare variabili, funzioni e tipi in *Tiger* è usare il costrutto LET-IN-END. **CUP**, al momento del riconoscimento dei token che compongono il costrutto, restituisce un oggetto della classe `AbSyn.LetExp` contenente tutte le dichiarazioni e le espressioni associate. La classe `Semant` chiama `transExp(AbSyn.LetExp)` apre un nuovo scope e processa le dichiarazioni. Finite le dichiarazioni processa le espressioni del corpo.

La parte più complicata riguarda le dichiarazioni di tipo o funzione ricorsive. L'analisi di queste dichiarazioni viene fatta in due momenti. Nel caso dei tipi, viene prima caricato nell'ambiente dei tipi il suo nome associandigli un tipo fittizio. Se si è dichiarata una funzione si inserisce nell'ambiente dei nomi senza processare il corpo, questo perchè potrebbe esservi usata la funzione appena dichiarata oppure una che verrà dichiarata successivamente. La seconda fase dell'analisi del `LetExp` lega ai tipi appena dichiarati il loro tipo reale, e per le funzioni processa il loro corpo. In caso di di dichiarazione di una funzione

viene creato un nuovo Livello contenente un nuovo frame e la lista dei parametri formali. Del Frame si parlerà dettagliatamente più in avanti.

BreakExp La classe `Semant` contiene una variabile di tipo `Translate .BreakScope`, che inizialmente è nulla. La classe `Translate.BreakScope` contiene la label che rappresenta il punto di fine ciclo. Ogni qualvolta viene trovato nell'albero di sintassi astratta un nodo `AbSyn.ForExp` o `AbSyn.WhileExp` viene creato un nuovo oggetto della classe `BreakScope`. Il corpo delle suddette espressioni viene analizzato creando un nuovo oggetto della classe `Semant` alla quale si passa `BreakScope`. L'oggetto `BreakExp` viene creato al momento del riconoscimento del token BREAK; se durante la fase dell'analisi semantica viene analizzata una `BreakExp` e l'oggetto `Translate.BreakScope` non esiste viene generato un'errore.

FunctionDec Quando viene dichiarata una funzione, viene creato un nuovo Livello. La classe `Translate.Level` è l'astrazione che permette al modulo `Semant` di usare i Frame senza tener conto dei dettagli implementativi che sono propri della machina obbiettivo. Contiene la lista dei parametri formali della funzione, rappresentati dagli *Accessi*. Questi sono l'astrazione delle locazioni di memoria e dei registri che andranno a contenere i valori delle variabili. Il Frame vero e proprio viene creato all'interno del Livello. Le variabili locali alle funzioni vengono create da `Frame`, che decide se allocare spazio in memoria o assegnargli un registro. La decisione si prende a seconda se la variabile *scappa* o no. Nel corpo delle funzioni si possono dichiarare variabili, chiamate *locali*. È anche possibile dichiarare funzioni. Se una di queste funzioni usa una di queste variabili, allora la variabile *scappa*. Semplicemente, il suo valore deve sopravvivere alla dichiarazione di una nuova funzione. `FindEscape` è il modulo che attraversa l'albero di sintassi astratta e segnala le variabili che scappano. Quando una variabile *scappa* il suo valore viene recuperato grazie allo *static link*. È una variabile all'interno del frame che contiene l'ndirizzo dell'inizio del frame precedente. Visto che lo *static link* è la prima informazioen di ogni Frame, è facile risalire ad ogni variabile dichiarata in un Frame precedente.

Confronto di Tipi Nel caso di confronto di due variabili di tipo RECORD, non è sufficiente confrontare che le due variabili siano dello stesso tipo. Questo perché ogni variabile di tipo RECORD può assumere il valore NIL. È necessario quindi permettere assegnamenti o confronti tra variabili di tipo RECORD e valore NIL. Per questo abbiamo scritto la funzione `checkSameType`.

```
boolean checkSameType (Type t1, Type t2){
  if (t1.actual() == t2.actual()) return true;
  if ((t1.actual() instanceof RECORD) && t2.actual() == NIL)
    return true;
  if ((t2.actual() instanceof RECORD) && t1.actual() == NIL)
    return true;
  return false;
}
```

Da notare l'uso di `actual()` ogni volta che ci si riferisce al tipo di una variabile, questo perché quando si definisce un nuovo tipo (*bind*) si lega il nome del

tipo (*NAME*) al tipo attuale. La funzione `actual()` ha il compito di restituire il tipo attuale di ogni variabile. Ad esempio

```
let
  type a = int
  type b = a
  type c = {x:d, y:c}
  type d = b
  var x:a := 0
  var y:c := c{x=3, y=nil}
  var z:d := 6
  var k:b := 10
in
  while (x+y.x)<(z+k) & y = nil do
    (
     y:=y.y;
     x:=x+1
    )
end
```

Questa espressione è legale.

Chiamata di Funzione Al momento della chiamata di funzione, oltre che controllare che la funzione sia stata precedentemente dichiarata, ci si assicura che i parametri attuali siano dello stesso tipo dei parametri formali. La traduzione in codice intermedio consiste in un nodo `Tree.CALL`. Le istruzioni che sistemano i parametri attuali nei registri [$a0-$a3] ed in memoria non sono visibili nel albero di codice intermedio, ma vongono generate al momento della traduzione del nodo dal generatore di codice. Invece vengono generate le istruzioni per prelevare i parametri attuali dai registri [$a0-$a3] e dalla memoria.

```
let
  var a:int :=99
  function f(x:int, y:int, z:int, k:int):int = a+x+y+z+k
in
  f(1,2,3,4)
end
```

```
f:
LABEL L2
MOVE(
 TEMP t31,
 MEM(
  BINOP(PLUS,
   TEMP $fp,
   CONST 8)))
MOVE(
 TEMP t30,
 TEMP $a3)
MOVE(
 TEMP t29,
 TEMP $a2)
MOVE(
 TEMP t28,
 TEMP $a1)
MOVE(
 MEM(
  TEMP $fp),
 TEMP $a0)
MOVE(
 TEMP $v1,
 BINOP(PLUS,
  BINOP(PLUS,
   BINOP(PLUS,
    BINOP(PLUS,
     MEM(
      BINOP(PLUS,
       CONST -4,
       MEM(
        TEMP $fp))),
     TEMP t28),
    TEMP t29),
   TEMP t30),
  TEMP t31))
JUMP(
 NAME L1)
LABEL L1
```

```
t_main:
LABEL L4
MOVE(
 MEM(
  BINOP(PLUS,
   CONST -4,
   TEMP $fp)),
 CONST 99)
MOVE(
 TEMP $v1,
 CALL(
  NAME f_0,
   TEMP $fp,
   CONST 1,
   CONST 2,
   CONST 3,
   CONST 4))
JUMP(
 NAME L3)
LABEL L3
```

Static Link Nel testo viene spiegato il concetto di *static link* di una funzione chiamata come l'indirizzo del frame di attivazione più recente della funzione nella quale è stata dichiarata. Per fronteggiare questa esigenza è stato necessario scrivere un metodo in `Translate.Translate` che calcolasse questa informazione, *distinta* dal metodo usato per calcolare i caricamenti dalla memoria nel caso delle variabili accessibili dall'interno di una funzione dichiarata nel corpo di un'altra funzione. Per mostrare il corretto funzionamento dell'implementazione forniamo come esempio l'esempio usato nel testo e la sua traduzione in codice intermedio:

```
let
```

```
  type tree={key:string, left:tree, right:tree}
  var output:=""
  var b := tree{key="APPEL", left=nil, right=nil}
  var a := tree{key="CIAO", left=b, right=b}

  function prettyprint(tree:tree) =
  let
    function write(s:string) = output:=concat(output, s)
    function show(n:int, t:tree) =
      let
        function indent(s:string) =
        (for i:=0 to n do
        write(" ");
        output:=concat(output, s);
        write("\n"))
      in
        if t = nil then indent(".")
        else (indent(t.key); show(n+1, t.left); show(n+1, t.right))
      end
   in
     show(0, a); output
   end
in
  prettyprint(a)
end
```

Mostriamo solo i nodi delle chiamate e degli usi delle variabili:

```
prettyprint
chiama show
passandogli il
suo frame
pointer come
static link:
MOVE(
TEMP $v1,
CALL(
 NAME show,
  TEMP $fp,
  CONST 0,
  TEMP t28))
```

```
show chiama
indent
passandogli il
suo frame
pointer come
static link:
EXP(
 CALL(
  NAME indent,
   TEMP $fp,
   MEM(
    BINOP(PLUS,
     TEMP t35,
     CONST 0)))
```

```
show chiama
show
passandogli il
suo static
link come
static link:
EXP(
 CALL(
  NAME show,
   MEM(
    TEMP $fp),
```

```
 indent usa la
 variabile n
 di show:
 CJUMP(LT,
  TEMP t38,
  MEM(
   BINOP(PLUS,
    CONST -4,
    MEM(
     TEMP $fp))),
  L5,L2)
```

```
indent chiama
write
passandogli il
frame pointer
di prettyprint
come static
link:
EXP(
 CALL(
  NAME write,
   MEM(
    MEM(
     TEM $fp)),
   NAME L3))
```

indent usa la variabile di prettyprint *output*

```
MOVE(
 TEMP t64,
 BINOP(PLUS,
  CONST -4,
  MEM(
   MEM(
    TEM $fp))))
```

Variabli, Record e Array Le variabili che si possono dichiarare in *Tiger* sono di tipo Array, Record o semplici(stringhe o interi). La fase dell'analisi semantica controlla che la variabile sia dichiarata, ed al momento dell'assegnazione che i due tipi combacino.

Per le variabili semplici, al momento della dichiarazione, viene riservato un registro o viene allocata della memoria (scappa o non scappa). Al momento del suo utilizzo viene cosi restituito il codice intermedio corrispondente alla sua posizione all'interno del Frame. Per la variabile di tipo Array viene allocato spazio in memoria pari alla dimensione per la grandezza di una *word* dell'architettura obbiettivo. Per accedere alla posizione i-esima di una variabile x di tipo Array, bisogna usare la sintassi "x[i]". Questo viene tradotto in un accesso alla locazione di memoria situata i*$word$ dopo la locazione della variabile x.

Per la variabile di tipo Record viene allocato spazio in memoria pari al numero di campi per la grandezza di una *word*. Per accedere al campo a che sta nella posizione i-esima di un record x, bisogna usare nella sintassi "x.a". Questo viene tradotto in un accesso alla locazione di memoria situata i*$word$ dopo la locazione iniziale della variabile x.

É possibile dichiarare anche Array di variabili record, vediamo un esempio:

```
let
  type rec = {a:int, tail:rec}
  type arr = array of rec
  var x:rec := rec{a=0, tail=nil}
  var y:arr := arr[10] of x
in
  for i:=1 to 9 do
  let
    var z:rec := rec{a=i, tail = y[i-1]}
  in
    y[i] := z
  end;
  x := y[9];
  while x<>nil do
    x:=x.tail
end
```

```
LABEL L7
MOVE(
 TEMP t28,
 CALL(
  NAME allocRecord,
   CONST 8))
MOVE(
 MEM(
  BINOP(PLUS,
   TEMP t28,
   CONST 0)),
 CONST 0)
MOVE(
 MEM(
  BINOP(PLUS,
   TEMP t28,
   CONST 4)),
 CONST 0)
MOVE(
 TEMP t30,
 TEMP t28)
MOVE(
 TEMP t31,
 CALL(
  NAME initArray,
   CONST 10,
   TEMP t30))
MOVE(
 TEMP t32,
 CONST 1)
LABEL L1
CJUMP(LE,
 TEMP t32,
 CONST 9,
 L2,L0)
LABEL L0
MOVE(
 TEMP t30,
 MEM(
  BINOP(PLUS,
   TEMP t31,
   BINOP(LSHIFT,
    CONST 9,
    CONST 2))))
LABEL L4
CJUMP(NE,
 TEMP t30,
 CONST 0,
 L5,L3)
LABEL L3
MOVE(
 TEMP $v1,
 CONST 0)
JUMP(
 NAME L6)
```

```
LABEL L2
MOVE(
 TEMP t33,
 CALL(
  NAME allocRecord,
   CONST 8))
MOVE(
 MEM(
  BINOP(PLUS,
   TEMP t33,
   CONST 0)),
 TEMP t32)
MOVE(
 MEM(
  BINOP(PLUS,
   TEMP t33,
   CONST 4)),
 MEM(
  BINOP(PLUS,
   TEMP t31,
   BINOP(LSHIFT,
    BINOP(MINUS,
     TEMP t32,
     CONST 1),
    CONST 2))))
MOVE(
 TEMP t34,
 TEMP t33)
MOVE(
 MEM(
  BINOP(PLUS,
   TEMP t31,
   BINOP(LSHIFT,
    TEMP t32,
    CONST 2))),
 TEMP t34)
MOVE(
 TEMP t32,
 BINOP(PLUS,
  TEMP t32,
  CONST 1))
JUMP(
 NAME L1)
LABEL L5
MOVE(
 TEMP t30,
 MEM(
  BINOP(PLUS,
   TEMP t30,
   CONST 4)))
JUMP(
 NAME L4)
LABEL L6

.
```

AssignExp e ForExp Per quanto riguarda l'istruzione *For*, é da notare che nella sintassi *Tiger* non é possibile riassegnare il valore della variabile di controllo nelle istruzioni interne al ciclo. Per fare questo, abbiamo introdotto un campo *modify* legato ad ogni variabile nell'ambiente (modificabile, non modificabile) e

un campo *force* legato all'istruzione AssignExp della sintassi astratta. L'assegnamento di una variabile sará cosi autorizzato se la variabile é "modificabile" o se l'istruzione di assegnamento forza la restrizione. Cosi nel caso di un'istruzione *For*, la variabile di controllo sará impostata "non modificabile" e solamente all'assegnamento incrementale sará permesso di forzare questa restrizione.

FOR-DO e WHILE-DO L'espressione FOR-DO viene trasformata in WHI -LE-DO dall'analisi semantica creando un nuovo albero. La traduzione è identica per entrambi i costrutti. Vediamo un esempio che mostra la trasformazione ed il metodo che la mette in atto.

```
let
var a:=0
var i:=0
in
for i:= 0 to 10 do
a:=a+i

while i<10 do
(a:=a+1; i:=i+1)
end
```

```
MOVE(
 TEMP $v1,
 ESEQ(
  SEQ(
   MOVE(
    TEMP t29,
    CONST 0),
   MOVE(
    TEMP t30,
    CONST 0)),
  ESEQ(
   SEQ(
    MOVE(
     TEMP t31,
     CONST 0),
    SEQ(
     JUMP(
      NAME L1),
     SEQ(
      LABEL L2,
      SEQ(
       SEQ(
        MOVE(
         TEMP t29,
         BINOP(PLUS,
          TEMP t29,
          TEMP t31)),
        MOVE(
         TEMP t31,
         BINOP(PLUS,
          TEMP t31,
          CONST 1))),
       SEQ(
        LABEL L1,
        SEQ(
         CJUMP(LE,
          TEMP t31,
          CONST 10,
          L2,L0),
         LABEL L0)))))),    .
```

```
ESEQ(
 SEQ(
  JUMP(
   NAME L4),
  SEQ(
   LABEL L5,
   SEQ(
    SEQ(
     MOVE(
      TEMP t29,
      BINOP(PLUS,
       TEMP t29,
       CONST 1)),
     MOVE(
      TEMP t30,
      BINOP(PLUS,
       TEMP t30,
       CONST 1))),
    SEQ(
     LABEL L4,
     SEQ(
      CJUMP(LT,
       TEMP t30,
       CONST 10,
       L5,L3),
      LABEL L3))))),
 CONST 0))))
```

```
ExpTy transExp(Level level, Absyn.ForExp e){
  ExpTy ret;
  Exp decl, cond;
  BreakScope breakscope = new BreakScope();
  Semant new_semant = new Semant(env, err, trans, level, breakscope);

  env.venv.beginScope();
  decl = new_semant.transDec(level, e.var, false);

  Absyn.Var i = new Absyn.SimpleVar(e.var.pos,e.var.name);

  cond = new_semant.transExp(level,
                             new Absyn.OpExp(e.pos,
                                             new Absyn.VarExp(e.var.pos, i),
                                             Absyn.OpExp.LE, e.hi)).exp;
  ret = new_semant.transExp(level,
            new Absyn.SeqExp(e.body.pos,
            new Absyn.ExpList(e.body,
              new Absyn.ExpList( new Absyn.AssignExp(e.hi.pos,
              i,
              new Absyn.OpExp(e.hi.pos,
                new Absyn.VarExp(e.hi.pos, i),
                Absyn.OpExp.PLUS,
                new Absyn.IntExp(e.hi.pos, 1) ), true),
                null) )));
  env.venv.endScope();
  Exp qqq = trans.seqExp(decl, trans.whileExp(cond, ret.exp, breakscope));
  return new ExpTy(qqq, VOID);
}
```

3 Dal codice intermedio al codice SPIM

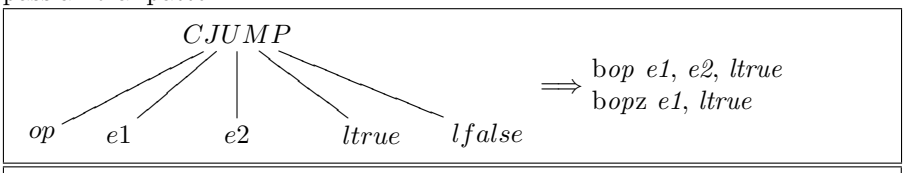
3.1 Generazione del Codice

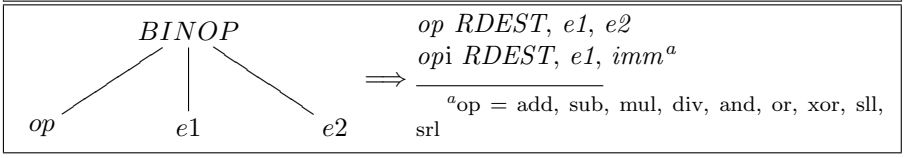
Il modulo `Mips.Codegen` genera un insieme di istruzioni a partire dal codice intermedio di ogni procedura o funzione, ottimizzato dal modulo `Canon`. Esso elimina i tipi di nodi `Tree.ESEQ` e `Tree.SEQ`, ed i salti ad etichette immediatamente successive, ottenendo una lista di sottoalberi.

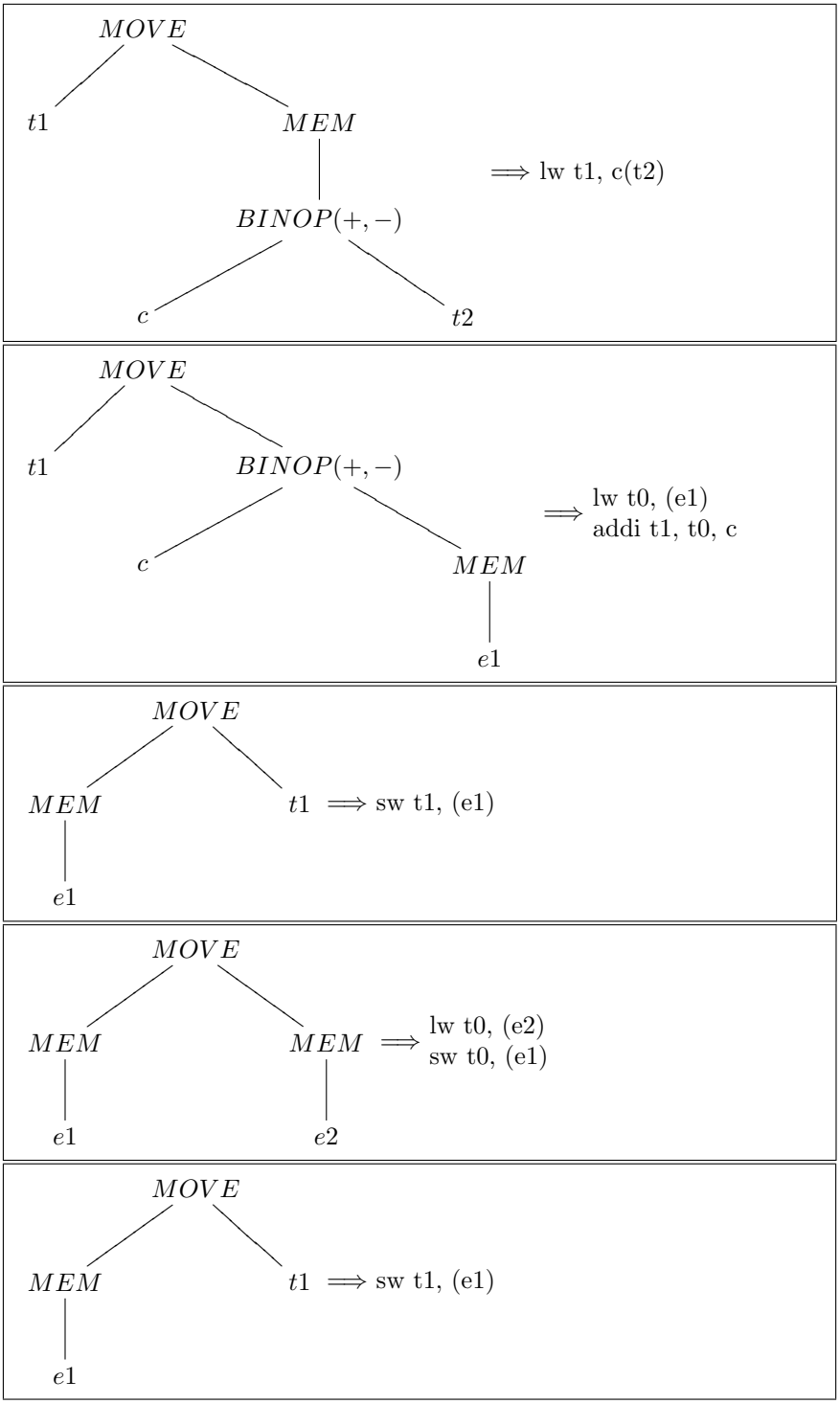
`Mips.Codegen` genera per ognuno di questi sottoalberi la corrispondente istruzione assembler. L'algoritmo usato si chiama **MaximalMunch**, e riconosce il tassello più grande possibile a partire dalla radice dell'albero. È un algoritmo di tassellamento ottimale. Due funziioni ricorsive, `munchStm` e `munchExp`, scorrono l'albero producendo le istruzioni corrispondenti ai tasselli. Prima di vedere i pattern più importanti osserviamo che nella scelta delle istruzioni sono state incluse anche alcune *pseudo istruzioni* di SPIM, in particolare:

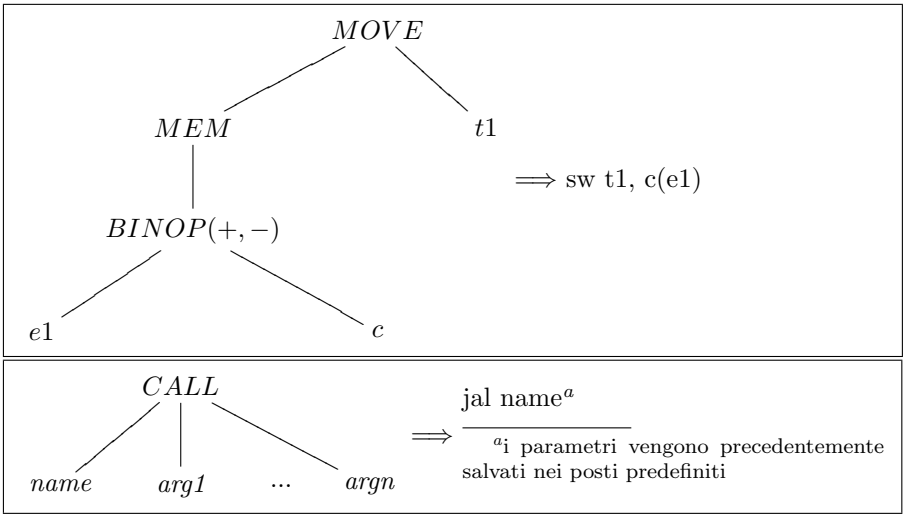
- `li Rdest, imm`
- `la Rdest, address`
- `mul Rdest, Rsrc1, Src2`
- `div Rdest, Rsrc1, Src2`
- `beqz Rsrc, label`
- `bge Rsrc1, Rsrc2, label`
- `bgt Rsrc1, Rsrc2, label`
- `ble Rsrc1, Rsrc2, label`
- `blt Rsrc1, Rsrc2, label`
- `bnez Rsrc, label`
- `move Rdst, Rsrc`

Aumetare la flessibilità del codice prodotto significherebbe generare codice MIPS "puro", cosa che renderebbe l'implementazione più complicata e sicuramente più istruttiva. Abbiamo scelto di non continuare per mancanza di tempo. Ora passiamo ai pattern:






MOVE
t1
MEM
BINOP(+, −)
c
t2
⟹ lw t1, c(t2)
MOVE
t1
BINOP(+, −)
c
MEM
e1
⟹ lw t0, (e1)
addi t1, t0, c
MOVE
MEM
e1
t1
⟹ sw t1, (e1)
MOVE
MEM
e1
MEM
e2
⟹ lw t0, (e2)
sw t0, (e1)
MOVE
MEM
e1
t1
⟹ sw t1, (e1)



Traduzione del Tree.CALL Prima di eseguire il comando di chiamata a funzione (`jal`) bisogna caricare i parametri in modo tale che la funzione chiamata possa recuperarli. A questo proposito ricordiamo che, come anche suggeriva Appel, il calcolatore MIPS e di conseguenza l'emulatore SPIM, ha 4 registri speciali riservati per salvare i primi 4 parametri delle funzioni. Il resto dei parametri vengono salvati in memoria, ad un offset da un puntatore noto alla funzione chiamata. In particolare, questi parametri vengono salvati ad un offset positivo dallo **stack pointer**, che diventerà il **frame pointer** della funzione chiamata grazie all'operazione di allocazione di un nuovo frame.

ProcEntryExit1, 2 e 3 Al momento della traduzione del corpo di una funzione in codice intermedio non vengono considerati alcuni aspetti importanti che riguardano il *prologo* e l'*epilogo* della funzione. In particolare non esistono le istruzioni per il prelievo dei parametri attuali, dell'allocazione delle stringhe nell'area **dati** e dell'allocazione del nuovo frame. Queste operazioni vengono fatte rispettivamente dalle funzioni `procEntryExit1`, `procEntryExit2` e `procEntryExit3`.

ProcEntryExit1 Si occupa di caricare i parametri attuali dai registri appositi e, se necessario, dalla memoria. L'area di memoria che li contiene si trova, come precedentemente spiegato, nei byte superiori al **frame pointer**.

ProcEntryExit2 Crea lo spazio per le stringhe usate nella funzione usando le direttive del emulatore SPIM. Successivamente aggiunge un'istruzione `nop` che ha la funzione di impedire l'uso da parte dell'allocatore dei registri di alcunio registri speciali come lo **stack pointer** ed il registro `$zero`.

ProcEntryExit3 Si occupa dell'allocazione di un nuovo frame. Sposta il **frame pointer** al posto dello **stack pointer** e decrementa quest'ultimo di *framesize*. La grandezza del frame viene calcolata a secondo del numero di *store* contante dall'allocatore dei registri ottimizzata da una considerazione sulle *store* effettuate per salvare in memoria i parametri delle funzioni con piú di tre

parametri. Infatti per ogni chiamata di funzione verranno utilizzate le stesse locazioni di memoria ed é per questo che la grandezza del frame viene calcolata considerando il numero di *store* che non riguardano i parametri delle funzioni incrementato dal numero di parametri formali salvati in memoria dalla funzione con piú parametri.

3.2 Liveness Analysis

La *liveness analysis* consiste nel analizzare le variabili usate(registri immaginari) nel codice prodotto e registrare il loro campo di azione, cioè per quali istruzioni il loro valore deve essere "vivo". A questo scopo come prima cosa si costruisce il grafo del flusso di dati. Ogni istruzione viene messa in un nodo e le due istruzioni successive vengono collegate da un arco diretto. Usando questo grafo si fa la *liveness analysis* vera e propria, registrando per ogni nodo l'insieme di variabili che sono "vive" all'entrata ed all'uscita. Queste informazioni servono alla fase successiva, l'**Allocazione dei Registri**.

L'algoritmo che crea il grafo diretto delle istruzioni prende in ingresso una lista di istruzioni e restituisce il grafo corrispondente, ed opera in due fasi. In una prima fase vengono creati gli archi tra i nodi che non rappresentano istruzioni di salto[1]. Nella seconda fase vengono collegati i nodi delle istruzioni di salto con le loro destinazioni. Per fare ciò ogni istruzione di tipo `Assem.LABEL`, che rappresenta un potenziale punto di salto, viene messa in una `HashTable`, e si mette in corrsipondenza con una lista di istruzioni che hanno come obbiettivo quella etichetta. Una volta finito il promo passaggio, ogni `Assem.LABEL` nella tabella viene collegata con ognuno dei nodi delle istruzioni associate.

Un caso, invece, non gestito di salto incondizionato è la chiamata a funzione. Essendo di fatto impossibile collegarere l'istruzione della chiamata a funzione ed il salto di ritorno, non si aggiungono archi tra questi nodi. Questo perchè ogni procedura o funzione del codice viene processata separatamente. L'istruzione di salto al punto successivo alla chiamata a funzione viene incluso nell'epilogo della funzione. Segue il codice dell'algoritmo.

```
file:FlowGraph/AssemFlowGraph.java
private void mkGraph(InstrList i)
{
  Node curr = newNode();
  Node next = null;
  HashMap labeltable = new HashMap();
  java.util.Stack callstack = new java.util.Stack();

  while(i!=null)
  {
    nodetoinstr.put(curr, i.head);
    instrtonode.put(i.head, curr);
    if(i.tail != null)
    {
      next = newNode();
      if(!isJump(i.head))
      {
        addEdge(curr, next);
      }

    }
    if(i.head instanceof Assem.LABEL)
      Assem.LABEL lab = (Assem.LABEL)i.head;
      if(!labeltable.containsKey( lab.label))
        labeltable.put( lab.label, new InstrList(i.head, null));
      else
      {
        InstrList b = (InstrList)labeltable.get(lab.label);
        InstrList a = InstrList.cat(b, new InstrList(i.head, null));
        labeltable.put(lab.label,a);
      }
      if(i.head.jumps()!=null)
        for(Temp.LabelList l=i.head.jumps().labels;l!=null;l=l.tail)
        {
          InstrList a = (InstrList)labeltable.get(l.head);
```

[1] condizionato ed incondizionato

```
          labeltable.put(l.head, new InstrList(i.head, a));
        }
      i=i.tail;
      curr=next;
    }
    java.util.Iterator iter = labeltable.keySet().iterator();
    while(iter.hasNext())
    {
      Temp.Label l = (Temp.Label)iter.next();
      InstrList a = (InstrList)labeltable.get(l);
      Node b = (Node)instrtonode.get(getLast(a));
      if(!(instr(b) instanceof Assem.LABEL))
        continue;
      for(; a.tail!=null; a=a.tail)
        addEdge((Node)instrtonode.get(a.head), b);
    }
}
```

Basic Blocks I nodi del grafo ottenuto sono in numero pari alle istruzioni della procedura. L'analisi del raggio di azione delle variabili ha un'ordine di complessita pari ad $O(N^4)$ nel caso peggiore. L'algoritmo si può ottimizzare in due modi: ordinando il grafo delle istruzioni e comprimendo le istruzioni che hanno un solo arco entrante ed un solo arco uscente in un *macronodo*. La prima ottimizzazione si giustifica dal fatto che l'algoritmo per la *liveness analysis* crea gli insiemi di variabili "vive" usando informazioni dai nodi successivi, che avrà quando processerà quei nodi. La seconda ottimizzazione crea un nuovo grafo al solo fine di velocizzare l'analisi delle variabili. Infatti, la fase dell'allocazione dei registri ricomporrà le informazioni originali.

L'algoritmo **BasicBlocks** (trovato su Aho Ullmann etc.) trova i nodi *leader*. Essi possono essere:

- Nodi senza archi entranti
- Nodi che sono obbiettivi di salto condizionato o incondizionato.
- Nodi che hanno un arco entrante da un nodo che rappresenta un salto condizionato o incondizionato.

Un macronodo conterrà, quindi, i nodi leader e tutti i nodi successivi fino ed escluso il prossimo nodo leader. L'implementazione:

```
file:FlowGraph/BasicBlocksAssemFlowGraph.java
public void basicBlocks()
{
  java.util.LinkedList l=new java.util.LinkedList();
  java.util.HashMap inverse = new java.util.HashMap();
  NodeList nodes = nodes();
  Node n = null;
  Node last = null;
  targets(l);
  for(; nodes != null; nodes = nodes.tail)
  {
    if(isLeader(nodes.head) || l.contains(nodes.head))
    {
      if(n != null)
        n.setSucc(last.succ());
      n = nodes.head;
      macronodes.put(n, new NodeList(n, null));
    }
    else
    {
```

```
        NodeList a = (NodeList)macronodes.get(n);
        NodeList b = new NodeList(nodes.head, null);
        macronodes.put(n, NodeList.append(a, b));
        last = nodes.head;
        mynodes = delete(nodes.head, mynodes);
        inverse.put(nodes.head, n);
      }
      if(nodes.tail == null)
        n.setSucc(last.succ());
    }
    for(nodes = nodes(); nodes !=null; nodes=nodes.tail)
    {
      for(NodeList a = nodes.head.pred(); a!=null; a=a.tail)
      if(!macronodes.containsKey(a.head))
      {
        rmEdge(a.head, nodes.head);
        Node headnode = (Node)inverse.get(a.head);
        addEdge(headnode, nodes.head);
        nodes.head.setPred(NodeList.append(nodes.head.pred(),
                           new NodeList(headnode, null)));
      }
      for(NodeList a = nodes.head.succ(); a!=null; a=a.tail)
        if(!macronodes.containsKey(a.head))
          rmEdge(nodes.head, a.head);
    }
}

private void targets(java.util.LinkedList l)
{
    for(NodeList a = nodes(); a != null; a = a.tail)
    {
      if(instr(a.head) instanceof Assem.JUMP ||
         instr(a.head) instanceof Assem.BRANCH ||
         instr(a.head) instanceof Assem.CALL)
        for(NodeList b = a.head.succ() ; b != null; b = b.tail)
          l.add(b.head);
    }
}

private boolean isLeader(Node n)
{
    if(n.pred() == null)
      return true;
    if(instr(n.pred().head) instanceof Assem.JUMP ||
       instr(n.pred().head) instanceof Assem.BRANCH ||
       instr(n.pred().head) instanceof Assem.CALL)
      return true;
    return false;
}
```

La complessità dell'algoritmo, dopo le due ottimizzazioni, è ridotta a $O(N)-O(N^2)$. Le equazioni però devono essere modificate per fuzionare correttamente sul nuovo grafo. Inizialmente le equazioni che definiscono gli insieme di variabili "vive" all'entrata ed all'uscita del nodo n sono:

$$in[n] = use[n] \cup (out[n] - def[n]) \quad (1)$$

$$out[n] = \bigcup_{s \in succ[n]} in[s] \quad (2)$$

Consideriamo i nodi n e p, con p unico successore di n ed n unico predeces-

sore di p. Le equazioni (1) e (2) per il nodo n diventeranno:

$$\begin{aligned} in[n] &= use[n] \cup (out[n] - def[n]) && (3)\\ out[n] &= \bigcup_{s \in succ[n]} in[s] = in[p] && (4)\\ in[p] &= use[p] \cup (out[p] - def[p]) && (5)\\ out[p] &= \bigcup_{s \in succ[p]} in[s] && (6) \end{aligned}$$

Così per l'equazione (3) sostituendo l'equazione (5):

$$in[n] = use[n] \cup (in[p] - def[n]) = use[n] \cup ((use[p] \cup (out[p] - def[p])) - def[n])$$

Considerando le identità $(A \cup B) - C = (A - C) \cup (B - C)$ e $(A - B) - C = A - (B \cup C)$ abbiamo:

$$in[n] = use(n) \cup ((use(p) - def(n)) \cup ((out(p) - def(p)) - def(n)))$$

$$in[n] = use(n) \cup ((use(p) - def(n)) \cup ((out(p) - (def(p) \cup def(n)))))$$

Otteniamo le due definizioni per gli insiemi d'uso e di definizione di due nodi:

$$\begin{aligned} use(np) &= use(n) \cup (use(p) - def(n)) && (7)\\ def(np) &= def(n) \cup def(p) && (8) \end{aligned}$$

Liveness Il metodo `Liveness` analizza il grafo modificato da `BasicBlocksAssemFlowGrarph`. Gli insiemi delle variabili sono rappresentati da l'interfaccia fornita da Appel, `Temp.TempList`. Li sono stati definiti i metodi per l'unione, la diffenrenza ed alre operazioni.

```
file:RegAlloc/Liveness.java
public void liveness(){
  int N = f.getNodeCount();
  TempList in1;
  TempList out1;
  do{
    for(int i=0; i<N; i++){
      in1 = TempList.copy(in[i].getSet());
      out1 = TempList.copy(out[i].getSet());
      Node n = in[i].getNode();
      in[i].setSet(TempList.union(f.use(n),
                   TempList.diff(out[i].getSet(), f.def(n))));
      TempList temp = null;
      for(NodeList k=out[i].getNode().succ();k!=null;k=k.tail)
      {
        temp = TempList.union(temp,
                              in[returnPos(k.head)].getSet());
      }
      out[i].setSet(temp);
      if(TempList.equals(in[i].getSet(),in1) &&
         TempList.equals(out[i].getSet(),out1))
      {
        done[i] = '0';
      }
```

```
      else
      {
        done[i] = '1';
      }
    }
  }
  while(!allZeros(done));
}
```

3.3 Allocazione dei Registri

L'allocazione dei registri è stata creata seguendo l'algoritmo fornito da Appel. Abbiamo implementato l'algoritmo tenedo conto anche delle funzioni di `coalesce` e `freeze`.

Il modulo `RegAlloc` si occupa di trovare una soluzione al problema di assegnazione delle variabili ai registri reali della macchina. A questo proposito usa le informazioni della **liveness analysis** precentemente documentata. È doveroso mettere in luce alcuni dettagli che ci aiuteranno durante l'illustrazione dell'implementazione del suddetto algoritmo. Durante questa sezione le parole "registro fittizio" e "variabile" avranno lo stesso significato, registri usati nella generazione del codice e facenti parte della "macchina con registri infiniti" precedentemente accennata.

- L'insieme dei registri usati è più piccolo di quello presente in una macchina MIPS e, di conseguenza, dell'emulatore SPIM. La differenza sta nell'uso dei registri `hi` e `lo`[2] che servono per le operazioni di moltiplicazione e divisione. Questi due registri contengono rispettivamente i 32 bit più significativi e meno significativi delle due operazioni. L'istruzione della moltiplicazione e della divisione risultano essere *pseudoistruzioni*[3]. Non avendo avuto quindi il bisogno di generare le istruzioni originali per il processore MIPS non abbiamo avuto la necessità di usare le istruzioni che gestiscono questi registri e quindi li abbiamo esclusi dall'insieme dei registri usabili.

- I registri sono divisi in quattro insiemi: i registri temporanei($t0, $t9), i registri che devono soppravvivere alle chiamate a funzione($s0, $s7), i registri risrvati per il passaggio dei parametri attuali($a0, $a3) ed i registri "speciali"($fp, $sp, $zero, $ra, $v0, $v1).

- Il numero di colori che l'allocatore dei registri ha a disposizione è pari al 18 ($s0-$s7 e $t0-$t9).

- L'allocatore dei registri ha bisogno di sapere quali sono i registri "precolorati" e qual'è l'insieme delle variabili che deve sistemare nei registri reali. Quelli precolorati sono i 28 registri macchina e l'insieme dei registri iniziale è l'insieme delle variabili (istanze della classe `Temp.Temp`) presenti nel programma.

- La classe `Liveness` tiene conto dei spostamenti di dati da un registro all'altro(`RegAlloc.cloalesce()` ha bisogno di queste informazioni).

Le strutture dati usate per l'implementazione dell'algoritmo sono quelle fornite da Appel, le stesse usate nel compilatore fino a questo istatante, ovvero `Temp.TempList, Node.NodeList` e `RegAlloc.MoveList`. Le rimanenti strutture dati usate sono fornite dalla libreria standard di Java. Per l'esecuzione dell'algoritmo abbiamo pensato di implementare un suggerimento fornito da Appel, e cioè associare ad ogni insieme di registri un intero. Ogni volta che una delle variabili cambia insieme gli viene assegnato l'intero associato al nuovo insieme. L'interfaccia usata è:

[2] In più ci sono i registri `at`, `k0`, `k1` e `gp` che non vengono usati perché riservati al sistema

[3] Il comportamento dell'istruzione di moltiplicazione "pura" `mul Rdest, Rsrc` si limita a spostare il contenuto del registro `lo` nel registro di destinazione

```
private java.util.Dictionary in=new java.util.Hashtable();
private NodeList add(NodeList nl, Node n, int i)
{
  if(in(n, i))
    return nl;
  in.put(n, new Integer(i));
  return new NodeList(n, nl);
}

private boolean in(Node n, int i)
{
  Integer where = (Integer)in.get(n);
  if(where == null)
    return false;
  return (where.intValue() == i);
}
```

Un'interfaccia analoga è stata creata per il grado delle variabili per i colori disponibili e per l'insieme dei nodi adiacenti ad ogni nodo. Le tabelle hash `degree` e `color` associano ad ogni nodo il suo grado nel grafo delle interferenze, ed il colore associatogli dal metodo `assignColors()` rispettivamente.

file:RegAlloc/RegAlloc.java

```
private java.util.Dictionary adjList = new java.util.Hashtable();
private java.util.Dictionary degree = new java.util.Hashtable();
private java.util.Dictionary color  = new java.util.Hashtable();


private int getDegree(Node n)
{
  return ((Integer)degree.get(n)).intValue();
}

private void putDegree(Node n, int i)
{
  degree.put(n, new Integer(i));
}

private NodeList getAdjList(Node n)
{
  return (NodeList)adjList.get(n);
}

private int getColor(Node n)
{
  Integer c=(Integer)color.get(n);
  return c.intValue();
}

private void putColor(Node n, int i)
{
  color.put(n, new Integer(i));
}
```

L'esecuzione dell'allocazione dei registri avviene in questo modo:

- il corpo tradotto in codice viene passato come parametro all'allocatore dei registri
- l'allocatore dei registri fa la **liveness analysis** del codice chiamando il metodo `liveness()` della classe`Liveness`

- crea il grafo delle interferenze iniziale contenente solamente i registri macchina (grafo completo) chiamando il metodo `precolored()`
- crea il grafo delle interferenze risultante alla **liveness analysis** chiamando `liveness.interference()`
- crea la lista delle adiacenze per ogni nodo chiamando `mkAdjList()`
- carica la classe `Color` che si occupa di applicare l'algoritmo di colorazione del grafo con i sovramenzionati parametri iniziali.

In finale, osserviamo come il metodo `makeWorklist()` oltre che creare gli insiemi di nodi iniziale associa un colore ad ogniuno dei registri precolorati e registra il grado di ogni nodo:

```
file: RegAlloc/RegAlloc.java metodo: makeWorklist

private InterferenceGraph init_ig;

private void makeWorklist()
{
  for(NodeList init=init_ig.nodes();init!=null;init=init.tail)
  {
    Temp.TempList a = ((AFrame.Frame)init_tmap).all();
    Temp.Temp b = init_ig.gtemp(init.head);
    putDegree(init.head, NodeList.len(getAdjList(init.head)));
    if(Temp.TempList.isIn(b, a))
    {
      for(tl = init_all_regs,i = 0; tl != null; tl = tl.tail,i++)
        if(tl.head == init_ig.gtemp(init.head))
          break;
      putColor(init.head, i);
      in.put(init.head, new Integer(PRECOLORED));
      continue;
    }

    ...
  }
}
```

Colorazione del grafo L'algoritmo di allocazione dei registri viene ripetuto finchè non ci sono più registri da "spillare", cioè da salvare e caricare in memoria della definizione e prima dell'uso, rispettivamente, del loro contenuto. Questa operazione viene effetetuata con l'ausilio di una nuova classe che rappresenta una nuova locazione di memoria: `Mips.MemoryLocation`

```
package AFrame;

public abstract class MemoryLocation
{
  abstract public Assem.InstrList fetch(Temp.Temp t);
  abstract public Assem.InstrList store(Temp.Temp t);
}

package Mips;

public class MipsMemoryLocation extends AFrame.MemoryLocation
{
  int offset;
```

```
  Frame f;

  public MipsMemoryLocation(int offset, Frame f)
  {
    this.offset = offset;
    this.f =f;
  }

  public Assem.InstrList fetch(Temp.Temp t)
  {
    String s = "lw `d0, " + offset + "($fp)\n";
    TempList a = new Temp.TempList(t,null);
    return new Assem.InstrList(new Assem.OPER(s, a, null), null);
  }

  public Assem.InstrList store(Temp.Temp t)
  {
    String s = "sw `s0, " + offset + "($fp)\n";
    TempList a = new Temp.TempList(t,null);
    return new Assem.InstrList(new Assem.STORE(s, null, a), null);
  }

}
```

Finita l'allocazione dei registri, per ogni registro il contenuto del quale deve essere messo in memoria si crea una nuova `Mips.MipsMemoryLocation`. Sostituiamo il registro da "spillare", nelle istruzioni che lo usano o lo definiscono, con un registro nuovo nel quale viene caricato il valore della locazione di memoria associata *prima* di ogni uso di esso ed il contenuto del quale viene salvato nella locazione di memoria *dopo* la definizione del registro originale. Queste due istruzioni vengono appese nella lista di istruzioni dal metodo `Regalloc.rewriteProgram()`.

```
private Assem.InstrList rewriteProgram(Assem.InstrList il)
{
  Temp.Temp best = color.getBest();
  AFrame.MemoryLocation loc = frame.allocMemoryLocation();
  il = eliminateTemp(il, best, loc);
  return il;
}
```

La funzione `eliminateTemp(Assem.InstrList il, Temp.Temp t, AFrame.MemoryLocation loc)` ha il compito di aggiungere queste istruzioni nel corpo della funzione. L'algoritmo viene riapplicato sul nuovo corpo della funzione.

Quando non ci sono più registri da spillare allora viene chiamata la funzione `assignColor()`, che fa uso della tabella hash `color` contenente l'associazione *nodo* $\rightarrow$ *intero*. Itera su una lista di interi che rappresenta i colori disponbili ed associa ad ogni registro immaginario un registro reale.

3.4 Runtime.spim e considerazioni finali

Dopo l'allocazione dei registri otteniamo un programma eseguibile nell'emulatore SPIM. Runtime.spim è il file contenente le funzioni di libreria precaricate nell'ambiente iniziale di ogni programma e le funzioni esterne di creazione di un nuovo `record` e di un `array`. In più, prima di ogni esecuzione, vengono caricati in un area del segmento dati gli interi $[0, 255]$, utili per la funzione di libreria `chr(i:int)`.

Le stringhe che compaiono nei programmi vengono create con la direttiva del simulatore `.asciiz` e sono automaticamente terminate da uno zero.

Per concludere riportiamo un'implementazione dell'algoritmo Mergesort che usa le funzioni di merge.tig, fornito da Appel, e ne riportiamo il codice SPIM prodotto.

```
let
  type any = {any : int}
  type list = {first: int, rest: list}
  var buffer := getchar()
  var list1:list := readlist()

  function readint(any: any) : int =
    let var i := 0
      function isdigit(s : string) : int =
        ord(s)>=ord("0") & ord(s)<=ord("9")
      function skipto() =
        while buffer=" " | buffer="\n"
           do buffer := getchar()
    in skipto();
       any.any := isdigit(buffer);
       while isdigit(buffer)
          do (
            i := i*10+ord(buffer)-ord("0");
            buffer := getchar()
          );
       i
    end

 function readlist() : list =
    let var any := any{any=0}
        var i := readint(any)
     in
        if any.any
        then list{first=i,rest=readlist()}
        else nil
    end
```

```
function merge(a: list , b: list) : list =
  if a=nil then b
  else if b=nil then a
  else if a.first < b.first
     then list{first=a.first ,rest=merge(a.rest ,b)}
     else list{first=b.first ,rest=merge(a,b.rest)}

function printint(i: int) =
 let function f(i:int) =
     if i>0 then
       (f(i/10); print(chr(i-i/10*10+ord("0"))))
 in if i<0 then (print("-"); f(-i))
    else if i>0 then f(i)
    else print("0")
 end

function printlist(l: list) =
  if l=nil then print("\n")
  else (printint(l.first); print(" ");printlist(l.rest))

function mergesort(l:list):list =
  let
    var l1:list := nil
    var l2:list := nil
    var num := 0
  in
    l1 := l;
    while l1<>nil do(
      num := num + 1;
      l1 :=l1.rest);
    if num=1 then
      l
    else(
         num := num / 2;
         l1 := l;
         for ii := 1 to num do(
          l2 := l1;
          l1 := l1.rest);
         l2.rest := nil;
         merge(mergesort(l), mergesort(l1))
         )
   end
in
 printlist(mergesort(list1))
end
```

```
isdigit_6:
move $fp, $sp
addi $sp, $sp, -60
.text
L67:
sw $a1, -8($fp)
sw $a0, ($fp)
move $a0, $fp
lw $a1, -8($fp)
sw $ra, -28($fp)
sw $fp, 4($sp)
jal ord
lw $ra, -28($fp)
sw $v1, -32($fp)
move $a0, $fp
la $a1, L8
sw $ra, -16($fp)
sw $fp, 4($sp)
jal ord
lw $ra, -16($fp)
lw $t9, -32($fp)
bge $t9, $v1, L10
L11:
li $s7, 0
L12:
move $v1, $s7
j L66
L10:
li $t9, 1
sw $t9, -36($fp)
move $a0, $fp
lw $a1, -8($fp)
sw $ra, -20($fp)
sw $fp, 4($sp)
jal ord
lw $ra, -20($fp)
sw $v1, -24($fp)
move $a0, $fp
la $a1, L9
sw $ra, -12($fp)
sw $fp, 4($sp)
jal ord
lw $ra, -12($fp)
lw $s7, -24($fp)
ble $s7, $v1, L13
L14:
li $s7, 0
sw $s7, -36($fp)
L13:
lw $s7, -36($fp)
j L12
L66:
nop
addi $sp, $sp, 60
lw $fp, 4($sp)
j $ra
.
```

```
skipto_7:
move $fp, $sp
addi $sp, $sp, -40

.text
L69:
sw $a0, ($fp)
L21:
li $t9, -4
lw $t8, ($fp)
lw $t8, ($t8)
add $t9, $t9, $t8
lw $a0, ($t9)
la $a1, L16
li $a2, 4
sw $ra, -16($fp)
sw $fp, 4($sp)
jal string
lw $ra, -16($fp)
bnez $v1, L18
L19:
li $t9, -4
lw $t8, ($fp)
lw $t8, ($t8)
add $t9, $t9, $t8
lw $a0, ($t9)
la $a1, L17
li $a2, 4
sw $ra, -20($fp)
sw $fp, 4($sp)
jal string
lw $ra, -20($fp)
bnez $v1, L22
L15:
li $v1, 0
j L68
L22:
lw $t9, ($fp)
lw $t9, ($t9)
addi $t9, $t9, -4
sw $t9, -12($fp)
move $a0, $fp
sw $ra, -8($fp)
sw $fp, 4($sp)
jal getchar
lw $ra, -8($fp)
lw $t9, -12($fp)
sw $v1, ($t9)
j L21
L18:
j L22
L68:
nop
addi $sp, $sp, 40
lw $fp, 4($sp)
j $ra
.
```

```
readint_0:
move $fp, $sp
addi $sp, $sp, -92

.text
L71:
sw $a1, -32($fp)
sw $a0, ($fp)
li $t9, 0
sw $t9, -52($fp)
move $a0, $fp
sw $ra, -12($fp)
sw $fp, 4($sp)
jal skipto_7
lw $ra, -12($fp)
lw $t9, -32($fp)
addi $t9, $t9, 0
sw $t9, -36($fp)
move $a0, $fp
li $t9, -4
lw $t8, ($fp)
add $t9, $t9, $t8
lw $a1, ($t9)
sw $ra, -16($fp)
sw $fp, 4($sp)
jal isdigit_6
lw $ra, -16($fp)
lw $t9, -36($fp)
sw $v1, ($t9)
L25:
move $a0, $fp
li $t9, -4
lw $t8, ($fp)
add $t9, $t9, $t8
lw $a1, ($t9)
sw $ra, -40($fp)
sw $fp, 4($sp)
jal isdigit_6
lw $ra, -40($fp)
bnez $v1, L26
L23:
lw $v1, -52($fp)
j L70
L26:
lw $t9, -52($fp)
mul $t9, $t9, 10
sw $t9, -44($fp)
move $a0, $fp
li $t9, -4
lw $t8, ($fp)
add $t9, $t9, $t8
lw $a1, ($t9)
sw $ra, -20($fp)
sw $fp, 4($sp)
jal ord
lw $ra, -20($fp)
lw $t9, -44($fp)
add $t9, $t9, $v1
sw $t9, -48($fp)
move $a0, $fp
la $a1, L24
sw $ra, -24($fp)
sw $fp, 4($sp)
jal ord
lw $ra, -24($fp)
lw $t9, -48($fp)
sub $t9, $t9, $v1
sw $t9, -52($fp)
lw $t9, ($fp)
addi $t9, $t9, -4
sw $t9, -28($fp)
move $a0, $fp
sw $ra, -8($fp)
sw $fp, 4($sp)
jal getchar
lw $ra, -8($fp)
lw $t9, -28($fp)
sw $v1, ($t9)
j L25
L70:
nop
addi $sp, $sp, 92
lw $fp, 4($sp)
j $ra
.
```

```
readlist_1:
move $fp, $sp
addi $sp, $sp, -68

.text
L73:
sw $a0, ($fp)
li $a0, 4
sw $ra, -24($fp)
sw $fp, 4($sp)
jal allocRecord
lw $ra, -24($fp)
li $t9, 0
sw $t9, 0($v1)
sw $v1, -28($fp)
lw $a0, ($fp)
lw $a1, -28($fp)
sw $ra, -16($fp)
sw $fp, 4($sp)
jal readint_0
lw $ra, -16($fp)
sw $v1, -8($fp)
lw $t9, -28($fp)
addi $t9, $t9, 0
lw $t9, ($t9)
li $t8, 0
bne $t9, $t8, L27
L28:
li $s7, 0
L29:
move $v1, $s7
j L72
L27:
li $a0, 8
sw $ra, -32($fp)
sw $fp, 4($sp)
jal allocRecord
lw $ra, -32($fp)
sw $v1, -36($fp)
lw $t9, -8($fp)
lw $t8, -36($fp)
sw $t9, 0($t8)
lw $t9, -36($fp)
addi $t9, $t9, 4
sw $t9, -20($fp)
lw $a0, ($fp)
sw $ra, -12($fp)
sw $fp, 4($sp)
jal readlist_1
lw $ra, -12($fp)
lw $s7, -20($fp)
sw $v1, ($s7)
lw $s7, -36($fp)
j L29
L72:
nop
addi $sp, $sp, 68
lw $fp, 4($sp)
j $ra
.
```

```
merge_2:
move $fp, $sp
addi $sp, $sp, -80

.text
L75:
sw $a2, -44($fp)
sw $a1, -40($fp)
sw $a0, ($fp)
lw $t9, -40($fp)
beqz $t9, L36
L37:
lw $t9, -44($fp)
beqz $t9, L33
L34:
lw $t9, -40($fp)
addi $t9, $t9, 0
lw $t9, ($t9)
lw $t8, -44($fp)
addi $t8, $t8, 0
lw $t8, ($t8)
blt $t9, $t8, L30
L31:
li $a0, 8
sw $ra, -28($fp)
sw $fp, 4($sp)
jal allocRecord
lw $ra, -28($fp)
sw $v1, -20($fp)
lw $t9, -44($fp)
addi $t9, $t9, 0
lw $t9, ($t9)
lw $t8, -20($fp)
sw $t9, 0($t8)
lw $t9, -20($fp)
addi $t9, $t9, 4
sw $t9, -12($fp)
lw $a0, ($fp)
lw $a1, -40($fp)
lw $t9, -44($fp)
addi $t9, $t9, 4
lw $a2, ($t9)
sw $ra, -36($fp)
sw $fp, 4($sp)
jal merge_2
lw $ra, -36($fp)
lw $s7, -12($fp)
sw $v1, ($s7)
lw $s7, -20($fp)
L32:
L35:
L38:
move $v1, $s7
j L74
L36:
lw $s7, -44($fp)
j L38
L33:
lw $s7, -40($fp)
j L35
L30:
li $a0, 8
sw $ra, -24($fp)
sw $fp, 4($sp)
jal allocRecord
lw $ra, -24($fp)
sw $v1, -16($fp)
lw $t9, -40($fp)
addi $t9, $t9, 0
lw $t9, ($t9)
lw $t8, -16($fp)
sw $t9, 0($t8)
lw $t9, -16($fp)
addi $t9, $t9, 4
sw $t9, -8($fp)
lw $a0, ($fp)
lw $t9, -40($fp)
addi $t9, $t9, 4
lw $a1, ($t9)
lw $a2, -44($fp)
sw $ra, -32($fp)
sw $fp, 4($sp)
jal merge_2
lw $ra, -32($fp)
lw $s7, -8($fp)
sw $v1, ($s7)
lw $s7, -16($fp)
j L32
L74:
nop
addi $sp, $sp, 80
lw $fp, 4($sp)
j $ra
```

```
f_39:
move $fp, $sp
addi $sp, $sp, -56

.text
L77:
sw $a1, -36($fp)
sw $a0, ($fp)
lw $t9, -36($fp)
bgtz $t9, L41
L42:
li $v1, 0
L43:
j L76
L41:
lw $a0, ($fp)
lw $t9, -36($fp)
div $a1, $t9, 10
sw $ra, -16($fp)
sw $fp, 4($sp)
jal f_39
lw $ra, -16($fp)
sw $fp, -8($fp)
sw $fp, -12($fp)
lw $t9, -36($fp)
div $t9, $t9, 10
li $t8, 10
mul $t9, $t9, $t8
lw $t8, -36($fp)
sub $t9, $t8, $t9
sw $t9, -24($fp)
move $a0, $fp
la $a1, L40
sw $ra, -20($fp)
sw $fp, 4($sp)
jal ord
lw $ra, -20($fp)
lw $a0, -12($fp)
lw $t9, -24($fp)
add $a1, $t9, $v1
sw $ra, -28($fp)
sw $fp, 4($sp)
jal chr
lw $ra, -28($fp)
lw $a0, -8($fp)
move $a1, $v1
sw $ra, -32($fp)
sw $fp, 4($sp)
jal print
lw $ra, -32($fp)
j L43
L76:
nop
addi $sp, $sp, 56
lw $fp, 4($sp)
j $ra
```

```
printint_3:
move $fp, $sp
addi $sp, $sp, -44

.text
L79:
sw $a1, -24($fp)
sw $a0, ($fp)
lw $t9, -24($fp)
bltz $t9, L49
L50:
lw $t9, -24($fp)
bgtz $t9, L46
L47:
move $a0, $fp
la $a1, L45
sw $ra, -20($fp)
sw $fp, 4($sp)
jal print
lw $ra, -20($fp)
L48:
L51:
j L78
L49:
move $a0, $fp
la $a1, L44
sw $ra, -8($fp)
sw $fp, 4($sp)
jal print
lw $ra, -8($fp)
move $a0, $fp
lw $t9, -24($fp)
sub $a1, $t9, 0
sw $ra, -12($fp)
sw $fp, 4($sp)
jal f_39
lw $ra, -12($fp)
j L51
L46:
move $a0, $fp
lw $a1, -24($fp)
sw $ra, -16($fp)
sw $fp, 4($sp)
jal f_39
lw $ra, -16($fp)
j L48
L78:
nop
addi $sp, $sp, 44
lw $fp, 4($sp)
j $ra
```

```
printlist_4:
move $fp, $sp
addi $sp, $sp, -44

.text
L81:
sw $a1, -24($fp)
sw $a0, ($fp)
lw $t9, -24($fp)
beqz $t9, L54
L55:
lw $a0, ($fp)
lw $t9, -24($fp)
addi $t9, $t9, 0
lw $a1, ($t9)
sw $ra, -8($fp)
sw $fp, 4($sp)
jal printint_3
lw $ra, -8($fp)
move $a0, $fp
la $a1, L53
sw $ra, -12($fp)
sw $fp, 4($sp)
jal print
lw $ra, -12($fp)
lw $a0, ($fp)
lw $t9, -24($fp)
addi $t9, $t9, 4
lw $a1, ($t9)
sw $ra, -20($fp)
sw $fp, 4($sp)
jal printlist_4
lw $ra, -20($fp)
L56:
j L80
L54:
move $a0, $fp
la $a1, L52
sw $ra, -16($fp)
sw $fp, 4($sp)
jal print
lw $ra, -16($fp)
j L56
L80:
nop
addi $sp, $sp, 44
lw $fp, 4($sp)
j $ra
```

```
mergesort_5:
move $fp, $sp
addi $sp, $sp, -64

.text
L83:
sw $a0, ($fp)
li $t9, 0
sw $t9, -28($fp)
li $t8, 0
li $t9, 0
sw $a1, -28($fp)
L58:
lw $t7, -28($fp)
bnez $t7, L59
L57:
beq $t9, 1, L63
L64:
div $t9, $t9, 2
sw $a1, -28($fp)
li $t6, 1
L61:
ble $t6, $t9, L62
L60:
li $t9, 0
sw $t9, 4($t8)
lw $t9, ($fp)
sw $t9, -8($fp)
lw $a0, ($fp)
sw $ra, -16($fp)
sw $fp, 4($sp)
jal mergesort_5
lw $ra, -16($fp)
sw $v1, -12($fp)
lw $a0, ($fp)
lw $a1, -28($fp)
sw $ra, -20($fp)
sw $fp, 4($sp)
jal mergesort_5
lw $ra, -20($fp)
lw $a0, -8($fp)
lw $a1, -12($fp)
move $a2, $v1
sw $ra, -24($fp)
sw $fp, 4($sp)
jal merge_2
lw $ra, -24($fp)
move $a1, $v1
L65:
move $v1, $a1
j L82
L59:
addi $t9, $t9, 1
lw $t7, -28($fp)
lw $t7, 4($t7)
sw $t7, -28($fp)
j L58
L63:
j L65
L62:
lw $t8, -28($fp)
lw $t7, -28($fp)
lw $t7, 4($t7)
sw $t7, -28($fp)
addi $t6, $t6, 1
j L61
L82:
nop
addi $sp, $sp, 64
lw $fp, 4($sp)
j $ra
```

```
t_main:
move $fp, $sp
addi $sp, $sp, -48

.data
L8: .asciiz "0"
L9: .asciiz "9"
L16: .asciiz " "
L17: .asciiz "\n"
L24: .asciiz "0"
L40: .asciiz "0"
L44: .asciiz "-"
L45: .asciiz "0"
L52: .asciiz "\n"
L53: .asciiz " "
.text
L85:
addi $t9, $fp, -4
sw $t9, -20($fp)
move $a0, $fp
sw $ra, -12($fp)
sw $fp, 4($sp)
jal getchar
lw $ra, -12($fp)
lw $t9, -20($fp)
sw $v1, ($t9)
move $a0, $fp
sw $ra, -24($fp)
sw $fp, 4($sp)
jal readlist_1
lw $ra, -24($fp)
sw $fp, -28($fp)
move $a0, $fp
move $a1, $v1
sw $ra, -16($fp)
sw $fp, 4($sp)
jal mergesort_5
lw $ra, -16($fp)
lw $a0, -28($fp)
move $a1, $v1
sw $ra, -32($fp)
sw $fp, 4($sp)
jal printlist_4
lw $ra, -32($fp)
j L84
L84:
nop
addi $sp, $sp, 48
lw $fp, 4($sp)
j $ra
```