

LINGUAGGI DI PROGRAMMAZIONE

COSIMO OLIBONI - PROGETTO 2006/2007: MINI C

Scanner (**scanner.flex**)

- 1.1) *Preprocessing*
- 1.2) *Scanning*

- 2.1) *Precedenza degli operatori*
- 2.2) *Grammatica*
- 2.3) *Ambiguità e parser GLR*
- 2.4) *Error recovery*

Dag (**parser_dag.y**)

- 3.1) *Algoritmo di analisi delle espressioni*

Type checking (**parser_type.y**)

- 4.1) *Tabella dei simboli e funzione hash*
- 4.2) *Dichiarazioni tipo ed equivalenza*
- 4.3) *Variabili globali e locali*
- 4.4) *Variabili e warning*
- 4.5) *Prototipi e funzioni*
- 4.6) *Ricorsione di coda*
- 4.7) *Cast implicito e ottimizzazione*
- 4.8) *Statement*
- 4.9) *For e return*
- 4.10) *While, equivalenza e return*

Codice intermedio (**parser_code.y**)

- 5.1) *Array n-dimensionali*
- 5.2) *Operatori logici, relazionali, etichette e booleani*
- 5.3) *Blocchi condizionali ed etichette*
- 5.4) *Passaggio di parametri*
- 5.5) *Controlli a run-time*
- 5.6) *Macroistruzioni e microistruzioni*
- 5.7) *Sommario delle istruzioni per compilare*

Materiale utilizzato:

- ◆ Windows: flex 2.5.4, bison 2.1, Visual C++ 6.0, PcSpim 7.1
- ◆ Linux: flex 2.5.31, bison 2.0, gcc 4.0.2, Xspim 7.3

***Il materiale prodotto è frutto esclusivamente del mio lavoro personale.
Nessuna porzione di codice né algoritmo è stato copiato da fonti esterne al corso.***

1.1) PREPROCESSING

L'automa gestisce un sottoinsieme limitato delle direttive messe a disposizione dal preprocessore del linguaggio C standard, applicando anche la sola espansione (-e):

Inclusione di altri sorgenti:

```
#include      sorgente
```

- ◆ E' ammesso l'annidamento di inclusioni.

Definizione di espansioni testuali:

```
#define      ID   <testo opzionale>
```

- ◆ E' ammesso l'utilizzo di espansioni all'interno della definizione.
ex:

```
#define  CONST    12
#define  EXP      (CONST*2)
```
- ◆ Non è ammessa la ridefinizione dei token propri del linguaggio.
ex:

```
#define  int      NEW_INT
```
- ◆ Non sono ammesse direttive del preprocessore all'interno della definizione.
ex:

```
#define  DEF      #define
```
- ◆ Non sono ammessi loop.
ex:

```
#define  CONST    VAL
#define  VAL      EXP
#define  EXP      CONST
```

Rimozione di espansioni testuali:

```
#undef      ID
```

- ◆ E' ammessa la rimozione di un identificatore non definito.

Scanning condizionato:

```
#ifdef      ID
<testo opzionale: scartato se ID non è definito>
#switch<opzionale>
<testo opzionale: scartato se ID è definito>
...
#endif
```

- ◆ E' ammesso l'annidamento di condizioni.
- ◆ Si può aggiungere un numero a piacere di *switch*.
- ◆ E' disponibile la versione negata *#ifndef*.

1.2) SCANNING

L'automata suddivide i token propri del linguaggio in categorie:

Keyword

<i>typedef</i>	<i>while</i>	<i>printf</i>	<i>if</i>	<i>getF</i>
<i>struct</i>	<i>for</i>	<i>printI</i>	<i>elseif</i>	<i>getI</i>
<i>const</i>	<i>return</i>	<i>printC</i>	<i>else</i>	<i>getC</i>
<i>prints</i>	<i>printNL</i>			

Tipi nativi

<i>float</i>	<i>int</i>	<i>char</i>	<i>void</i>
--------------	------------	-------------	-------------

NB: Keyword e tipi nativi costituiscono **termini riservati** del linguaggio.

Parentesi

{ } [] ()

Operatori

! ~ >> << ++ -- & | ^ + -
* / % != == < > <= >= = && ||

Separatori

; . ,

Vengono inoltre riconosciute espressioni regolari per le seguenti categorie:

Numeri positivi in virgola mobile

$[0-9]^+ \backslash. [0-9]^+ ([eE] [|\+|-]? [0-9]^+)?$

Numeri positivi interi

$[0-9]^+$

Singoli caratteri

$\backslash' [^\backslash n] \backslash'$

Stringhe

$\backslash" [^\backslash n]^* \backslash"$

Identificatori

$[_a-zA-Z] [_a-zA-Z0-9]^*$

Commenti (scartati)

$"/^*" [^\backslash n]^* "*/"$

2.1) PRECedenza degli Operatori

<i>SIMBOLO</i>	<i>OPERAZIONE</i>	<i>TIPO</i>	<i>ASSOCIATIVITA'</i>
<i>(exp)</i>	Raggruppamento		
<i>exp ++</i> <i>exp --</i>	Incremento postfisso Decremento postfisso	Unario Unario	
<i>id (explst)</i> <i>id [exp]</i> <i>id . id</i>	Applicazione funzione Selezione array Selezione campo		Destra
<i>++ exp</i> <i>-- exp</i> <i>! exp</i> <i>~ exp</i> <i>- exp</i> <i>+ exp</i>	Incremento prefisso Decremento prefisso Not logico Not bit-a-bit Cambiamento segno	Unario Unario Unario Unario Unario Unario	
<i>(type) exp</i>	Type cast		
<i>exp * exp</i> <i>exp / exp</i> <i>exp % exp</i> <i>exp + exp</i> <i>exp - exp</i> <i>exp << exp</i> <i>exp >> exp</i>	Moltiplicazione Divisione Modulo Addizione Sottrazione Shift sinistro Shift destro	Binario Binario Binario Binario Binario Binario Binario	Sinistra Sinistra Sinistra Sinistra Sinistra Sinistra Sinistra
<i>exp < exp</i> <i>exp <= exp</i> <i>exp > exp</i> <i>exp >= exp</i> <i>exp == exp</i> <i>exp != exp</i>	Minore Minore o uguale Maggiore Maggiore o uguale Uguale Diverso	Binario Binario Binario Binario Binario Binario	
<i>exp & exp</i> <i>exp exp</i> <i>exp ^ exp</i>	And bit-a-bit Or bit-a-bit Xor bit-a-bit	Binario Binario Binario	Sinistra Sinistra Sinistra
<i>exp && exp</i> <i>exp exp</i>	And logico Or logico	Binario Binario	Sinistra Sinistra
<i>id = exp</i>	Assegnamento	Binario	
<i>exp , exp</i>	Separazione		Destra

2.2) GRAMMATICA

Il linguaggio MINI C si propone come sottoinsieme del C standard con variazioni.

I) **Prologo** -> anche vuoto:

- ◆ variabili globali: *costanti, array, inizializzate, non inizializzate.*
- ◆ definizione nuovi tipi: *alias, strutture.*
- ◆ prototipi di funzioni.

II) **Funzioni** -> almeno una della forma *void main(void)*

E' importante sottolineare che:

- ◆ dopo la prima funzione non è possibile inserire ulteriori elementi del prologo.
ex: *void main(void) { return; } typedef int ALIAS;*
- ◆ l'inizializzazione di una variabile globale non può contenere applicazioni.
ex: *int glob=init();*
- ◆ gli array vengono definiti come intervalli interi (dimensione staticamente nota).
ex: *int array[-2,2];*
- ◆ il tipo void può essere usato solo nei prototipi (data l'assenza di puntatori).
ex: *void func(int param); int func(void);*
- ◆ i prototipi e le funzioni non possono definire array come valore di ritorno.
ex: *int[-2,2] func(int arg);*
- ◆ in ogni caso una variabile non può essere dichiarata in base a se stessa.
ex: *int var=var+1;*

ed inoltre:

- ◆ le strutture e le variabili sono passate **per valore** (come ogni valore di ritorno).
- ◆ gli array sono passati **per riferimento** (utilizzo implicito di un puntatore).

I blocchi condizionali implementati sono:

- ◆ *while (<exp>) <body>*
- ◆ *if (<exp>) <body> elsif (<exp>) <body> ... else <body>*
- ◆ *for (<new int id> <range>) <body>*

Il range della variabile 'fresca' su cui itera il for (iteratore) può essere definito dinamicamente. Non è presente l'interruzione di flusso break.

Un blocco (<body>) mantiene sempre la stessa struttura:

- ◆ *{ <locals> <statements> }*

Vengono fornite anche direttive di input/output per emulare printf/scanf:

- ◆ out di stringa: *printS (<string>)* *printNL*
- ◆ out di valore: *printF(<exp>)* *printI(<exp>)* *printC(<exp>)*
- ◆ in di valore: *getF(<l-val>)* *getI(<l-val>)* *getC(<l-val>)*

2.3) AMBIGUITA' E PARSER GLR

L'automa prodotto da Bison riporta 14 conflitti shift/reduce e 1 reduce/reduce, dovuti ad alcune ambiguità intrinseche della grammatica C, incorporate in MINI C per maggiore aderenza al modello base. Questa scelta impone l'uso di un parser più potente del modello LALR(1) standard.

Il parser GLR (**Generalized LR**) risolve ogni ambiguità applicando un'analisi parallela di ogni possibile ramificazione. Durante questa fase le azioni semantiche vengono solo registrate. I rami corrispondenti ad una scelta errata termineranno in uno stato di errore e spariranno. L'ultimo ramo sopravvissuto effettuerà un commit della pila temporanea di azioni semantiche su quella vecchia.

La potenza del parser GLR permette inoltre di rendere opzionale la presenza del separatore “;” all'interno delle produzioni, emettendo un semplice warning.

Le seguenti modifiche consentirebbero l'uso di un semplice parser LALR(1):

- ◆ utilizzare un separatore fra prologo e funzioni.
ex: `int glob; '%%' void main(void) {}`
- ◆ utilizzare un separatore fra variabili locali e statements.
ex: `{ int glob; '%%' return(1); }`
- ◆ diversificare la sintassi del cast.
ex: `('cast' int) 2.0`
- ◆ rendere obbligatoria la presenza del separatore “;” quando richiesto.

2.4) ERROR RECOVERY

La grammatica include alcuni semplici casi di error recovery di errori sintattici, limitati a casi in cui l'errore sia **ben delimitato a sinistra e a destra**, per ridurre al minimo il numero di token scartati da bison e non aumentare il numero di conflitti:

```
<struct>      ->  typedef struct { error } <id>
<prototype>  ->  <type> <id> ( error )
<range>      ->  [ error ]
<body>       ->  { error }
<for>        ->  for ( error ) <body>
<exp>        ->  ( error )
```

Terminata con successo la costruzione di un AST coerente (nessun errore di sintassi) comincerà la fase di type checking, dove tutti gli errori context-dependent verranno risolti in loco e fatti galleggiare, consentendo la continuazione del parsing.

3.1) ALGORITMO DI ANALISI DELLE ESPRESSIONI

In qualsiasi punto del programma espressioni che coinvolgano solo valori di tipo nativo vengono immediatamente contratte al tipo più rappresentativo fra i due operandi (upcast):

- *float* *op* ... -> *float*
- *int* *op* ... -> *int*
- *char* *op* *char* -> *char*

In alcuni particolari punti dell'AST è possibile, sotto precise condizioni (1-3), sostituire i normali puntatori con una struttura DAG. Questa ottimizzazione determina, in tutte le fasi successive, l'abbandono dei puntatori di tipo child->father e next->prev, forzando l'uso di tecniche sostitutive per lo scambio di informazioni nei casi corrispondenti.

1 Le produzioni interessate sono:

- *for* (<id> [<exp> , <exp>]) <body>
- *const* <type> <id> = <exp>
- *if/elsif/while/return/printv* (<exp>) ...
- <id> = <exp>

2 La valutazione avviene da destra a sinistra, cioè secondo una visita posticipata.

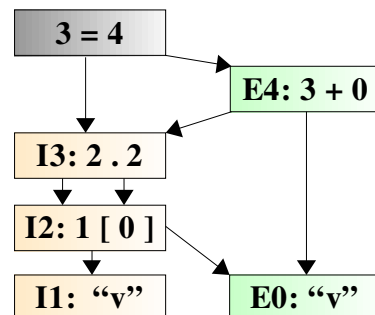
3 L'espressione (<exp>) non deve contenere chiamate a funzione (effetti collaterali).

Il normale meccanismo illustrato a lezione deve essere però potenziato (4-8) per gestire l'ambiguità campo/espressione e gli operatori di modifica prefissa/postfissa.

4 Una stringa può essere considerata come id o espressione a seconda del contesto.

$$\text{ex: } (\underline{v}^{Id\ 1} [\underline{v}^{Exp\ \theta}]^{Id\ 2} . \underline{v}^{Id\ 1} [\underline{v}^{Exp\ \theta}]^{Id\ 2})^{Id\ 3} = ((\underline{v}^{Id\ 1} [\underline{v}^{Exp\ \theta}]^{Id\ 2} . \underline{v}^{Id\ 1} [\underline{v}^{Exp\ \theta}]^{Id\ 2})^{Id\ 3} + \underline{v}^{Exp\ \theta})^{Exp\ 4} ;$$

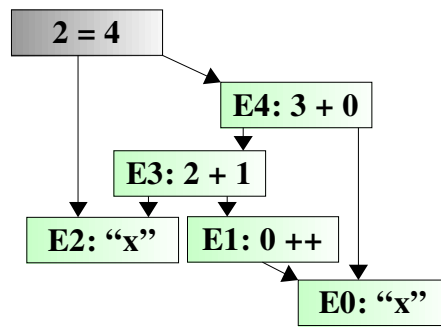
N.	Tipo		# Rif.	Inv.?
<u>0</u>	exp	"v"	5	no
<u>1</u>	id	"v"	4	no
2	id	1 [0]	4	no
3	id	2 . 2	2	no
4	exp	3 + 0	1	no



5 Una modifica postfixa applica alla sottoespressione la regola “aggiungi e invalida”.

ex: $\underline{x}^{Exp\ 2} = ((\underline{x}^{Exp\ 2} + (\underline{x}^{Exp\ 0}++)^{Exp\ 1})^{Exp\ 3} + \underline{x}^{Exp\ 0})^{Exp\ 4}$;

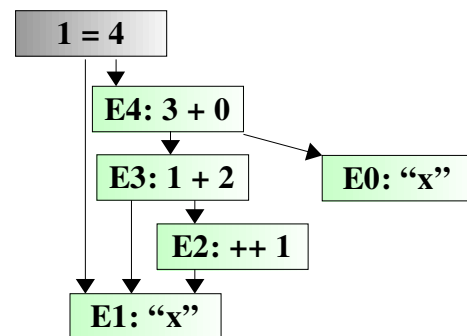
N.	Tipo		# Rif.	Inv.?
<u>0</u>	exp	“x”	2	si
1	exp	0 ++	1	si
<u>2</u>	exp	“x”	2	no
3	exp	2 + 1	1	si
4	exp	3 + 0	1	si



6 Una modifica prefissa applica alla sottoespressione la regola “invalida e aggiungi”.

ex: $\underline{x}^{Exp\ 1} = ((\underline{x}^{Exp\ 1} + (++\underline{x}^{Exp\ 1})^{Exp\ 2})^{Exp\ 3} + \underline{x}^{Exp\ 0})^{Exp\ 4}$;

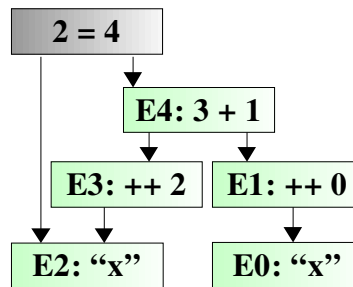
N.	Tipo		# Rif.	Inv.?
<u>0</u>	exp	“x”	1	si
<u>1</u>	exp	“x”	3	no
2	exp	++ 1	1	si
3	exp	1 + 2	1	si
4	exp	3 + 0	1	si



7 Una modifica prefissa o postfixa costituisce sempre un’espressione “invalidata”.

ex: $x^{Exp\ 2} = ((\underline{++x}^{Exp\ 2})^{Exp\ 3} + (\underline{++x}^{Exp\ 0})^{Exp\ 1})^{Exp\ 4}$;

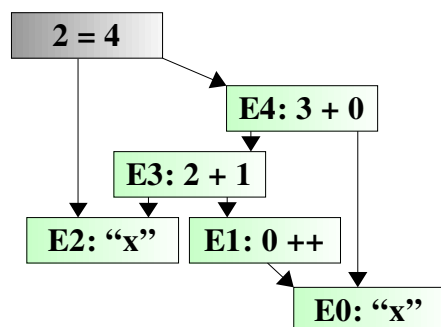
N.	Tipo		# Rif.	Inv.?
0	exp	“x”	1	si
<u>1</u>	exp	++ 0	1	si
2	exp	“x”	2	no
<u>3</u>	exp	++ 2	1	si
4	exp	3 + 1	1	si



8 Un nodo con un figlio invalidato risulta a sua volta invalidato.

ex: $x^{Exp\ 2} = ((x^{Exp\ 2} \pm (x^{Exp\ 0}++)^{Exp\ 1})^{Exp\ 3} \pm x^{Exp\ 0})^{Exp\ 4}$;

N.	Tipo		# Rif.	Inv.?
0	exp	“x”	2	si
1	exp	0 ++	1	si
2	exp	“x”	2	no
<u>3</u>	exp	2 + 1	1	si
<u>4</u>	exp	3 + 0	1	si



4.1) TABELLA DEI SIMBOLI E FUNZIONE HASH

E' stata creata una funzione hash flessibile che utilizza un algoritmo simile alla crittografia DES. Il risultato $\text{hash}(\text{name})$ sarà compreso tra 0 e $\text{MAX} \in [2^8-1, 2^{32}-1]$, a seconda di un unico parametro. I test denotano un ottimo grado di sparsità anche per nomi adiacenti (in base al valore ASCII dei caratteri):

ex: $[\text{MAX}=4095]: \text{hash}("a")=451, \text{hash}("b")=1123, \dots$

4.2) DICHIARAZIONI DI TIPO ED EQUIVALENZA

All'interno del prologo sono possibili definizioni di alias e strutture:

- ◆ *typedef <type> <ID>*
- ◆ *typedef struct { <field-list> } <ID>*

Queste dichiarazioni vengono inserite nella tabella dei simboli associate alla **categoria tipi**, calcolando immediatamente l'equivalenza con un tipo nativo o già definito. Gli alias vengono sempre considerati equivalenti al tipo precedente. Per le strutture vale l'equivalenza per nome e tipo dei campi, non necessariamente in ordine.

ex: *typedef struct { int a; char b; } s1;*
typedef struct { char b; int a; } s2;

Le dichiarazioni valutate equivalenti spariscono dall'AST e vengono memorizzate nella tabella dei simboli in modo da effettuare la sostituzione letterale all'interno dell'AST.

ex: *{ s1 var1; s2 var2; ... } -> { s1 var1; s1 var2; ... }*

Durante il calcolo delle equazioni di tipo si confronteranno quindi solo:

- ◆ tipi nativi (<0): *-float, -int, -char, -void, -error*
- ◆ nuovi tipi (≥ 0): *strutture non equivalenti*

Gli indici negativi (void ed error esclusi) indicheranno la possibilità di conversione in caso di necessità con upcast o downcast (viene emesso un warning). Gli indici positivi, facendo riferimento a strutture non equivalenti non ammetteranno possibilità di conversione. I tipi speciali void ed error non sono sottoposti ad equivalenza.

Da notare che fra i parametri formali sono ammessi array. Nel particolare caso di applicazioni di funzioni sono quindi possibili confronti fra array n-dimensionali che avvengono passando a parte un descrittore associato alle dichiarazioni del parametro formale e di quello attuale. Sono ammesse anche sezioni di array:

ex: *void f(int p[0,10]) { int v[-2,2][0,10]; f(v[0]); }*

4.3) VARIABILI GLOBALI E LOCALI

Le variabili (globali e locali) ammettono 4 forme di dichiarazione, inserite nella tabella dei simboli associate alla **categoria variabile**:

- ◆ non inizializzate: `<type> <ID>`
- ◆ inizializzate: `<type> <ID> = <exp>`
- ◆ costanti inizializzate: `const <type> <ID> = <exp>`
- ◆ array n-dimensionali: `<type> <ID> <range-list>`

Il costrutto `for` introduce anche una dichiarazione implicita di variabile intera per l'iteratore visibile all'interno del blocco:

```
ex:  for(index[exp1,exp2]) { ... }
```

Bisogna considerare i seguenti vincoli:

- non esistono puntatori, né variabili di tipo `void`.
- l'inizializzazione può essere applicata solamente ai tipi nativi.
- l'inizializzazione di variabili globali deve ridursi ad un valore costante.
- le costanti possono essere sostituite nell'AST solo se riducono a valori costanti.
- non devono esistere campi omonimi in una struttura.
- non devono esistere parametri formali omonimi in un prototipo/funzione.
- non devono esistere variabili omonime allo stesso livello di annidamento.

L'ultimo vincolo coinvolge anche omonimie che coinvolgano i parametri formali, l'iteratore del `for` e le variabili locali:

```
ex:  int func(int param) { float param; ... }  
ex:  for(index[exp1,exp2]) { float index; ... }
```

4.4) VARIABILI E WARNING

Al momento della distruzione di un ambiente, le variabili mai accedute provocano un warning, controllo molto semplice da implementare.

Molto più complicato si presenta il controllo di inizializzazione di una variabile. La presenza di array e strutture richiederebbe l'associazione di un descrittore di inizializzazione ad ogni membro della variabile. Per questo motivo il controllo non è stato implementato (in alcuni compilatori è presente), assumendo un'implicita inizializzazione a 0.

```
ex:  typedef struct { int field1; int field2; } myStruct; ...  
void func(void) {  
    int array[0,10]; myStruct var; ...  
    array[0]=1; var.field1=1;  
    if ((array[1]<??>+var.field2<??>)>2) { ... }  
    ... }
```

4.5) PROTOTIPI E FUNZIONI

Le funzioni, precedute da al massimo un unico prototipo, ammettono due forme di dichiarazione, inserite nella tabella dei simboli associate alla **categoria prototipo/funzione**:

- ◆ prototipo senza parametri: *<type> <ID> (void)*
- ◆ prototipo con parametri: *<type> <ID> (<param-list>)*
- ◆ funzione senza parametri: *<type> <ID> (void) <body>*
- ◆ funzione con parametri: *<type> <ID> (<param-list>) <body>*

L'utilizzo di prototipi che dichiarino il comportamento delle funzioni ancor prima delle loro definizioni permette:

- ◆ estrema libertà nell'ordine delle definizioni delle funzioni.
- ◆ mutua ricorsione.

La ripartizione in categorie distinte di tipi/variabili/funzioni all'interno della tabella dei simboli permette di scremare molte omonimie:

ex: *ID_{TYPE} ID_{FUNC}(ID_{TYPE} ID_{VAR}) { ... }*

Bisogna comunque considerare i seguenti vincoli:

- il tipo di ritorno non può essere un array.
- il body deve terminare con un return compatibile con il tipo dichiarato.

4.6) RICORSIONE DI CODA

Tenendo opportunamente traccia dei contenuti del body è possibile stabilire la possibilità/opportunità di abilitare la ricorsione di coda, alle seguenti condizioni:

- deve apparire almeno una chiamata ricorsiva.
- tutte le chiamate devono essere ricorsive, in coda, al di fuori di espressioni.

```
ex:  /* uso factTailRec(n>=0,1); */
      int factTailRec(int num,int parz)
      {
        if(num<=1) { return(parz); }
        else { return(factTailRec(n-1,parz*n)); }
      }
```

```
/* uso factNoTailRec(n>=0); */
int factNoTailRec(int num)
{
  if(num<=1) { return(1); }
  else { return(n*factNoTailRec(n-1)); }
}
```

4.7) CAST IMPLICITO E OTTIMIZZAZIONE

Nella valutazione delle espressioni si segue la stessa regola descritta al 3.1, portando il tipo al più rappresentativo tra i due operandi coinvolti mediante inserimento di un cast nell'AST. Possono ancora tuttavia avvenire contrazioni, dovute alla sostituzione delle costanti con i valori espliciti di inizializzazione.

ex: $a\langle float \rangle + b\langle int \rangle \rightarrow a + (float) b$

ex: $const\ int\ c=2; int\ b=c+1; \rightarrow int\ b=3;$

Speciali espressioni vengono ottimizzate, sostituendo la valutazione nell'AST, nonostante coinvolgano anche variabili. Non devono in ogni caso apparire modificatori di ambiente quali applicazioni/incrementi/decrementi:

- operatori relazionali fra espressioni unificate a DAG.

ex: $ID == ID \rightarrow 1$ $(exp1 + exp2) != (exp1 + exp2) \rightarrow 0$

- sottrazione fra espressioni unificate a DAG.

ex: $ID - ID \rightarrow 0$ $(exp1 * exp2) - (exp1 * exp2) \rightarrow 0$

- divisione/moltiplicazione/somma/sottrazione per l'elemento neutro.

ex: $ID * 1 \rightarrow ID$ $ID - 0 \rightarrow ID$

- operatori logici fra espressioni unificate a DAG.

ex: $ID \&\& ID \rightarrow ID$ $ID || ID \rightarrow ID$

ex: $ID \&\& !ID \rightarrow false$ $ID || !ID \rightarrow true$

- operatori logici con operandi booleani immediati.

ex: $true \&\& ID \rightarrow ID$ $true || ID \rightarrow true$

ex: $false \&\& ID \rightarrow false$ $false || ID \rightarrow ID$

ex: $ID \&\& true \rightarrow ID$ $ID || true \rightarrow (invariato)$

ex: $ID \&\& false \rightarrow (invariato)$ $ID || false \rightarrow ID$

- ◆ la valutazione con cortocircuito sinistro viene preservata.

4.8) STATEMENT

Oltre ai blocchi condizionali ed ai comandi print/return, sono ammessi statement costituiti da espressioni con side effect (modifiche sull'ambiente):

- incremento/decremento postfisso/prefisso di un l-value.
- chiamate di procedure (tipo di ritorno void).
- assegnamenti.

Da notare che contrariamente al C

- scartare il valore di ritorno di una funzione costituisce errore.
- un assegnamento produce void come valore di ritorno.

Il primo caso viene controllato in fase di type checking, il secondo caso viene impedito a priori in fase di parsing.

4.9) FOR E RETURN

Il corpo di un for viene per forza eseguito almeno una volta. La presenza di un return o di un blocco equivalente al suo interno elimina quindi il codice seguente.

ex: `for(i[exp1,exp2]) { return; } <dead code> ...`

Alle due espressioni, la cui valutazione a run-time delimiterà l'iteratore, è consentito produrre qualsiasi risultato intero: `exp1 < exp2`, `exp1 == exp2`, `exp1 > exp2`.

4.10) IF, WHILE, EQUIVALENZA E RETURN

L'espressione collegata ad un if/elsif/while può essere di un qualsiasi tipo nativo, dove false è associato a 0 e true altrimenti.

While

Nel semplice caso del while, si distinguono tre casi:

- while(false) viene rimosso dall'AST.
- while(true) equivale ad un iteratore infinito ed elimina il codice seguente.
- while(exp) ha un comportamento imprevedibile.

La presenza di un return o di un blocco equivalente al suo interno verrà quindi considerata solo nel secondo caso.

ex: `while(true) { return; } <dead code> ...`

If

Si profilano diverse trasformazioni possibili:

- if(false) viene rimosso dall'AST:
 - l'else seguente viene trasformato in if(true).
 - l'elsif seguente viene trasformato in if.
- ex: `if(false) { } elsif(exp) { } -> if(exp) { }`
- elsif(false) viene rimosso dall'AST.
- if(true) rimuove dall'AST di tutti gli altri rami.
 - ex: `if(true) { } elsif(exp) { } -> if(true) { }`
- elsif(true) viene trasformato in else e rimuove dall'AST tutti gli altri rami.
 - ex: `if(exp) { } elsif(true) { } -> if(exp) { } else { }`
- if/elsif(exp) ha un comportamento imprevedibile.
- else completa qualsiasi insieme di diramazioni.

La presenza di un return o di un blocco equivalente all'interno di **ogni** corpo di un insieme completo di diramazioni elimina quindi il codice seguente.

```

ex:  if(exp)
      {
        if(exp)
          { return; }
        else
          { <missing return> }
      }
    else
      { return; }
    <alive code>
    ...

```

Da notare che in ogni caso fin qui elencato, il codice dichiarato morto non è mai sottoposto a controlli di type check (strategia lazy).

5.1) ARRAY n-DIMENSIONALI

Vengono utilizzate le seguenti formule, che minimizzano le operazioni da effettuare a run-time, per calcolare l'offset dell'elemento/sezione dell'array:

- $size_i = end_i - start_i + 1$
- $array[in_1]..[in_k] \rightarrow array[in_1]..[in_k][start_{k+1}]..[start_n]$
- **1 dimensione:** $base + \{in_1 - start_n\} * sizeof(elem)$
- **2 dimensioni:** $base + \{(in_2 - start_2) + size_2 * (in_1 - start_1)\} * sizeof(elem)$
- **n dimensioni:** $base + \{(in_n - start_n) + size_{n-1} * (.. * (in_1 - start_1))\} * sizeof(elem)$

La versione finale a n-dimensioni gode già della proprietà di poter essere calcolata in forma ricorsiva in esattamente n passi. Un ulteriore miglioramento deriva dal memorizzare lo spiazamento dovuto alla parte fissa $start_1..start_n$:

- **n dimensioni:** $base + \{const + [in_n + size_{n-1} * (.. * in_1)]\} * sizeof(elem)$

5.2) OPERATORI LOGICI, RELAZIONALI, ETICHETTE E BOOLEANI

Tutti gli operatori condizionali e relazionali producono:

- o un valore booleano 1/0 (true/false):
 - ex: $var = (x < y); \quad var = (x | | y);$
- o un salto ad un etichetta:
 - ex: $while(x < y) \{..\} \quad while(x | | y) \{..\}$

Per la valutazione di queste espressioni viene richiesta **un'unica visita all'AST**, che riesce a produrre, per mezzo di opportune ottimizzazioni un numero di macroistruzioni estremamente ridotto:

[exp1 AND exp2]

```
if false(e1) goto FALSE
if true(e2) goto TRUE
<jump FALSE>opt
```

[exp1 OR exp2]

```
if true(e1) goto TRUE
if true(e2) goto TRUE
<jump FALSE>opt
```

[exp1 NAND exp2]

```
if false(e1) goto TRUE
if false(e2) goto TRUE
<jump FALSE>opt
```

[exp1 NOR exp2]

```
if true(e1) goto FALSE
if false(e2) goto TRUE
<jump FALSE>opt
```

[exp1 RELOP exp2]_{jumpToTrue}

```
set $reg to (e1 RELOP e2)
branch $reg!=0, TRUE
```

[exp1 RELOP exp2]_{jumpToFalse}

```
set $reg to (e1 RELOP e2)
branch $reg==0, FALSE
```

[exp1 RELOP exp2]

```
set $reg to (e1 RELOP e2)
move newTemp, $reg
```

Le etichette TRUE/FALSE possono corrispondere a:

- un etichetta generata da un blocco condizionale.
- un etichetta ereditata da un'espressione logica padre.
- un etichetta generata da un'espressione logica radice.

FALSE:

```
move newTemp, 0
jump NEXT
```

TRUE:

```
move newTemp, 1
<jump NEXT>opt
```

L'istruzione *jump FALSE* che concluderebbe ogni operatore logico viene emessa solo nel caso di espressioni radice, per evitare macroistruzioni superflue del tipo:

```
100: jump Label2
101: Label2:
```

5.3) BLOCCHI CONDIZIONALI ED ETICHETTE

While(exp) { body }

```
NewLabel1:
    emit(exp, TRUE=NewLabel2, FALSE=NewLabel3, jumpToFalse);
NewLabel2:
    emit(body, NEXT=NewLabel1);
NewLabel3:
```

```

if(exp1) { body1 }
elseif(exp2) { body2 }
else {body3 }

```

```

    emit(exp1,TRUE=NewLabel2,FALSE=NewLabel3,jumpToFalse);
NewLabel2:
    emit(body1,NEXT=NewLabel6);
NewLabel3:
    emit(exp2,TRUE=NewLabel4,FALSE=NewLabel5,jumpToFalse);
NewLabel4:
    emit(body2,NEXT=NewLabel6);
NewLabel5:
    emit(body3,NEXT=NewLabel6);
NewLabel6:

```

```

for(ID[exp1,exp2]) { body }

```

```

    emit("ID[0]=exp1");
    emit("ID[1]=exp2");
    emit("ID[0]<=ID[1] ? ID[2]=+1 : ID[2]=-1");
    emit("ID[1]+=ID[2]");
    emit("jump NewLabel2");
NewLabel1:
    emit("ID[0]+=ID[2]");
    emit("branch ID[0]==ID[1] NewLabel3");
NewLabel2:
    emit(body,NEXT=NewLabel1);
NewLabel3:

```

5.4) PASSAGGIO DI PARAMETRI

Il passaggio di parametri passa al momento della chiamata dallo stack:

- float/int/char per valore: *pushFloat/Int/Char* <r-val>
- strutture per valore: *pushBlock* <l-val> <size>
- array per puntatore: *pushInt* <l-val>

Al momento del ritorno il risultato passa invece da un apposito registro, per ovviare alla possibile corruzione dello stack dovuta ad applicazioni multiple che precedano il prelievo del valore:

```

ex:  array[func1()][func(2)]=func3();

```

I risultati di tipo non nativo (strutture) vengono anch'essi restituiti per valore e ciò implica l'allocazione di memoria in cui copiare il risultato. Questi blocchi di memoria vengono allineati alla dimensione della struttura più grande definita all'interno del programma. L'inevitabile **frammentazione interna** è il prezzo da pagare per una semplice allocazione tramite **lista libera**.

```

struct s func1(void) { .. }
struct s func2(struct s param) { .. }
struct s var=func2(func1());

```

...	func2:	func1:
call func1
pushBlock retVal,size	mallocBlock	mallocBlock
freeBlock retVal	memCpy B,...,size	memCpy B,...,size
call func2	return B	return B
memCpy var,retVal,size		
freeBlock retVal		
...		

5.5) CONTROLLI A RUN-TIME

Vengono implementati i seguenti controlli a run-time:

- divisione per 0.
- Top-Of-Stack > Top-Of-Dinamic-Area
- indici di ogni array entro le dimensioni di inizio/fine dichiarate.

5.6) MACROISTRUZIONI E MICROISTRUZIONI

Per semplificare il processo di emissione del codice intermedio, vengono generate pseudo istruzioni in un assembly a 3 indirizzi, ricco di macroistruzioni potenti, modulari, prive di vincoli registro/memoria. Segue poi un'espansione in microistruzioni MIPS, secondo tutti i vincoli intero/double, registro/memoria, etc...

```

ex:  addint [gp+n],tInt,2    ->  move $reg3,2
                                   move $reg2,tInt
                                   add $reg1,$reg2,$reg3
                                   move [gp+n],$reg1

```

5.7) SOMMARIO DELLE ISTRUZIONI PER COMPILARE

Semplice espansione del preprocessore:

```
./scanner source > source.txt
```

Parsing con output di AST:

```
./scanner source | ./parser -v > source.txt
```

Compilazione senza inizializzazione delle variabili a 0:

```
./scanner source | ./parser -n > source.s
```

Compilazione con inizializzazione della variabili a 0:

```
./scanner source | ./parser > source.s
```