ISTITUTO
TECNICO
ARCHIMEDE

MyC

CONTINA LORENZO GIUSEPPE 5EINF

ANNO SCOLASTICO 2022/23

DESCRIZIONE DELL'IDEA PROGETTUALE E OBIETTIVI

Si presenta la realizzazione di un compilatore per un linguaggio della famiglia "C", ovvero con paradigma imperativo, procedurale e strutturato, e utilizzo generico.

Un linguaggio ad uso generico è in grado di risolvere un vasto dominio di problemi.
Implementando il modello computazionale della macchina di Turing, questi linguaggi sono teoricamente in grado di implementare qualsiasi algoritmo, e quindi qualsiasi programma.
Storicamente, I linguaggi della famiglia C sono utilizzati per risolvere un gran numero di problemi:

- Sistemi operativi: ad esempio Unix, che è strettamente legato alla nascita del linguaggio C;
- Videogiochi;
- Server WEB;
- Motori database;
- Software da ufficio;
- Sofware per sistemi integrati;
- ...

Il compilatore comprende 5 fasi, divise in "front-end" e "back-end":

Front-end:

1. Analisi lessicale (lexer): si legge Il codice sorgente come stringa, e si analizza il contesto di ogni carattere, producendo una lista di simboli.
2. Pre-processore (prepar): in questa fase si leggono alcune direttive che il compilatore deve interpretare prima dell'analisi sintattica. Le direttive sono definite dall'implementazione, non dal linguaggio. Possono comprendere operazioni di sostituzione, inserimento ed eliminazione di simboli.
3. Analisi sintattica (parser): si legge la lista di simboli e ne si comprende il significato nel loro contesto, ovvero la relazione tra i vari simboli. Questa fase produce una struttura ad albero chiamata AST (Abstract Syntax Tree, ovvero Albero di Sintassi Astratto).

Back-end:

1. Tipizzazione (typecheck): l'AST viene attraversato e vengono stabiliti I tipi di ogni espressione. Definisce le regole di tipizzazione del linguaggio,

ovvero: quali operazioni sono permesse e per quali tipi, l'uguaglianza di due tipi, la compatibilità tra tipi diversi.

2. Generazione del codice (codegen): l'AST tipizzato viene attraversato e per ogni nodo viene generato del codice in un linguaggio più semplice (in questo caso assembly x64), e un risultato (registro o regione di memoria).

Il progetto ha motivi prettamente autodidattici e di ricerca.
Infatti, il primo obiettivo del progetto è imparare a comprendere meglio come funzionano i compilatori e, più in generale, i computer.
Esperienze sul campo ci insegnano che Il linguaggio C è molto semplice, e lascia molto spazio al programmatore. Ciò può risultare in problemi di sicurezza legati all'acesso della memoria, alla concorrenza.
Il progetto, quindi, si pone come secondo obiettivo quello di risolvere alcuni problemi del linguaggio C.

Trattandosi di un progetto software, realizzato da una singola persona, non si stimano alcuni costi di realizzazione.

DESCRIZIONE DEL CONTENUTO INNOVATIVO DEL PROGETTO

La scelta di questo genere di linguaggio, piuttosto che un altro, è giustificabile dalla presenza di un gran margine di miglioramento nei linguaggi di questa famiglia.
Il linguaggio C soffre della mancanza di molte funzionalità e caratteristiche legate alla sicurezza e alla facilità d'uso, che MyC si prospetta di colmare:

- Limitare le operazioni aritmetiche su puntatori: un linguaggio sicuro non dovrebbe permettere operazioni aritmetiche sui puntatori.
- Boundary Checking: il controllo dei limiti quando si indirizza un array può aiutare a risolvere grossi problemi di sicurezza della memoria.
- Array/Stringa come puntatore e dimensione: immagazzinare la dimensione degli array e delle stringhe di caratteri per permettere il boundary checking, e migliorare le prestazioni.

La sintassi del linguaggio C è spesso considerata come "classica" o "normale" (soprattutto da programmatori inesperti o alle prime armi). Tuttavia a volte ci troviamo a scrivere espressioni o dichiarazioni che sono al limite dell'illegibilità.[immagine 1]

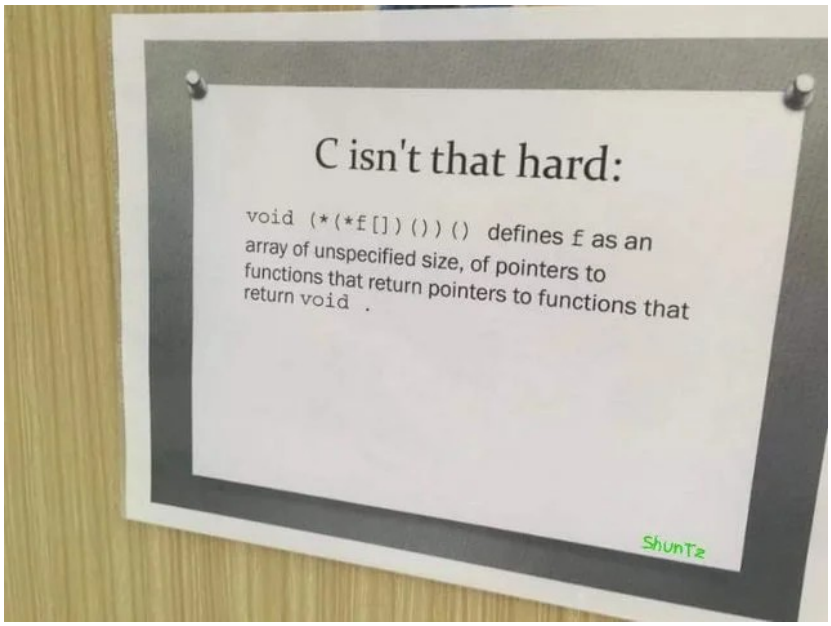


IMMAGINE 1: MEME SULLA SINTASSI DELLE DICHIARAZIONI COMPLESSE NEL LINGUAGGIO C.

MyC cerca di semplificare la grammatica delle dichiarazioni, seguendo una logica ben precisa, che prova a rispecchiare la logica dei linguaggi naturali (linguaggi “umani”).

FASI DI SVILUPPO DEL PROGETTO

1. Ricerca sulle tecnologie, i problemi, le soluzioni adottate da diversi linguaggi/compilatori;
2. Progettazione (design);
3. Sviluppo delle varie parti del compilatore;
4. Convalida o testing.

Queste non sono delle fasi sequenziali, ma concorrenti; infatti progettazione e ricerca, come sviluppo e testing, sono avvenuti comtemporaneamente.

COMPONENTI, STRUMENTI E ATTREZZATURE

Il compilatore è interamente scritto in Go. La scelta di questo linguaggio è del tutto arbitraria ed irrilevante ai fini del raggiungimento degli obiettivi del progetto.
L’assembler utilizzato è “gas”, ovvero l’assembler di GNU: presente nel pacchetto gcc, supporta la maggior parte dei sistemi Unix e Linux.

Per il testing e la ricerca ho utilizzato alcuni strumenti di debugging e analisi:

- gdb;
- compiler explorer (https://godbolt.org/).

ARCHITETTURA E DETTAGLI IMPLEMENTATIVI

Le montagne sono protagoniste di molte analogie; esiste una simile analogia anche per i compilatori:
quando ci troviamo ai piedi di una montagna, ci è impossibile vedere cosa sta dall'altro lato -- qui troviamo il codice in caratteri "crudi"; questo è quindi il punto dove abbiamo meno informazioni contestuali, e ci è difficile comprendere il codice abbastanza bene da poterlo tradurre;

quando raggiungiamo la cima della montagna abbiamo la vista libera, e riusciamo ad osservare bene tutto ciò che ci sta attorno -- qui troviamo l'AST tipizzato, che è la rappresentazione con più informazioni contestuali di sintassi e di semantica. Da qui è facile comprendere il codice e tradurlo in un linguaggio più semplice.

Di seguito sono analizzati dettagliatamente tutti i passaggi effettuati dal compilatore:

LEXER

L'analisi lessicale traduce una stringa "cruda" di caratteri in una lista di simboli.
Si tratta di un'analisi lineare, ovvero carattere per carattere, di tutto il codice.
Le regole lessicali definiscono I rapporti tra più caratteri, e quali simboli rappresentano.

Un simbolo è costituito da un tipo, un valore numerico o testuale, informazioni sulla posizione all'interno del codice sorgente (numero di riga e di carattere).
I tipi di simbolo sono suddivisi in 6 categorie: carattere, multi-carattere, parole chiave, valori letterali, identificatori, direttive.

```
:tried_letters [8]uint8;
```

Questa dichiarazione di array risulta nella seguente sequenza di simboli:

DUE_PUNTI,
IDENTIFICATORE("tried_letters"),

PARENTESI_QUADRA_APERTA,
LETTERALE_INTERO(8),
PARENTESI_QUADRA_CHIUSA,
IDENTIFICATORE(“uint8”),
PUNTO_E_VIRGOLA.

I valori letterali di stringa sono delimitati da doppi apici (`”`). I letterali di carattere sono delimitati da apici (`’`).
I caratteri bianchi, come spazi, tabulazioni, avanzamenti di riga, sono considerati separatori, e non risultano in alcun simbolo.
Qualsiasi carattere dopo il prefisso `//`, fino ad un avanzamento di riga, viene ignorato.
Qualsiasi carattere delimitato dai prefissi `/*` e `*/`, viene ignorato.
Le direttive sono simili ai commenti, e utilizzano questi caratteri: `@` per una singola riga, `@@` per più righe. Le direttive, a differenza dei commenti, producono un simbolo che viene successivamente interpretato dal pre-processore.

PARSER

L’analisi sintattica traduce una lista di simboli in un albero sintattico.
Questa è un’analisi ricorsiva, che costruisce l’albero dall’alto verso il basso.
Ogni nodo dell’albero contiene: tipo, flag, dati (lista di simboli), tipo di dato (questo campo viene determinato solo durante il passaggio successivo), figli.
Molti compilatori decidono di usare un albero binario (al massimo due figli per ogni nodo); questa implementazione, invece, permette un grado indefinito per ogni nodo.

DEFINIZIONI DI FUNZIONE

```
function name([dichiarazione,] [dichiarazione,] …) tipo {
	...
}
```

CORPI DI CODICE

```
{
	[espressione/dichiarazione/altro ;]
```

```
        …
        [espressione]
    }
```

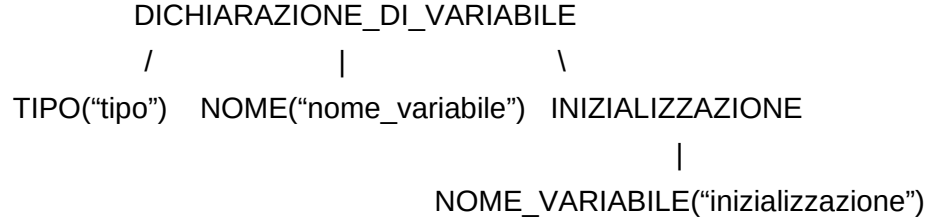
Le espressioni dentro un corpo di codice sono sempre delimitate da un punto e virgola, se non si tratta del risultato del corpo.
Ogni blocco di codice è un'espressione; questo ci permette di scrivere funzioni pulite:

```
    function type(:size uint64) uint8 {
        switch size {
            case 8:  {BYTE}
            case 16: {WORD}
            case 32: {DOUBLE}
            case 64: {QUAD}
        }
    }
```

DICHIARAZIONI

```
    :nome_variabile tipo [= inizializzazione];
```

In questo esempio, il parser leggerebbe il simbolo DUE_PUNTI, e chiamerebbe la funzione per analizzare una dichiarazione, che produrrebbe questo risultato:



ESPRESSIONI E OPERATORI

Esistono due tipi di operatori: binari, unitari.

Operatori binari:

```
4 * 5 + 3;
```

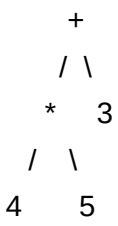
Questa espressione verrebbe analizzata come segue:

```
analizza_espressione():
        leggo 4,
        leggo *:
                *
               / \
              4
        analizza_espressione():
                leggo 5,
                leggo +:
                        +
                       / \
                      5
                analizza_espressione():
                        leggo 3,
                        leggo ;,
                        restituisco 3:

                restituisco
                        +
                       / \
                      5   3
        restituisco
                *
               / \
              4   +
                 / \
                5   3
```

È facilmente intuibile che questo risultato è sbagliato, secondo le regole sull'ordine degli operatori che ci impone la matematica.

Infatti l'operatore * ha una priorità più alta rispetto all'operatore +.
Per "aggiustare" quest'albero, effettuiamo un'operazione di rotazione a sinistra, che ci restituisce questo risultato:



Operatori unitari:

Sintatticamente, si distingue tra due classi di operatori unitari: di coda e di testa:

```
a[i]; // coda
a++;  // coda
++a;  // testa
```

Gli operatori di coda sono attesi dopo una *sottoespressione,* mentre quelli di testa sono attesi prima.
Entrambi questi operatori non prevedono l'aggiustamento dell'ordine degli operatori.

CHIAMATA A FUNZIONE

```
funzione([espressione,] [espressione,] …);
```

IF

```
if espressione {
     ...
} [ else {
     ...
} ]
```

WHILE

```
while espressione {
      ...
}
```

SWITCH

```
switch espressione {
      case espressione: {…}
      …
}
```

Le espressioni all'inizio di switch, while ed if, sono racchiuse implicitamente dentro due parentesi tonde, per risolvere l'ambiguità con i letterali composti.

FOR

```
for([dichiarazione]; [espressione]; [espressione]) {
      ...
}
```

JUMP

```
jump label;
...
label:
...
```

RETURN

```
return [espressione];
```

LETTERALE COMPOSTO

```
tipo {
      [espressione,]
      ...
      [espressione,]
```

```
}
```

DEFINIZIONE STRUTTURA

```
struct nome {
    [dichiarazione;]
    [dichiarazione;]
    ...
}
```

TIPI

```
a uint8;                // tipo primitivo
b *uint8;               // tipo puntatore
c [[DIMENSIONE]]uint8; // tipo array
```

TIPI

PRIMITIVI

Interi

```
int64
int32
int16
int8
uint64
uint32
uint16
 uint8
```

Altro

```
bool
```

```
? (puntatori/array generici)
```

PUNTATORI

I tipi puntatore sono definiti dall'utente, ed hanno un tipo elementare, e una dimensione che dipende dalla piattaforma.

I puntatori generici sono compatibili con tutti gli altri puntatori, e sono un'alternativa ai puntatori `void*` di C.

STRUTTURE

Le strutture sono definite dall'utente ed hanno:

- un nome;
- una dimensione;
- uno scopo dove si trovano le definizioni dei membri;
- una lista di membri (fields).

ARRAY

I tipi array sono definiti dall'utente.
Gli array statici hanno un tipo elementare ed una lunghezza.
La lunghezza è stabilita staticamente e non è modificabile (solo lettura).
La dimensione di questi tipi è data dal prodotto tra la dimensione dell'elemento e la lunghezza.

Gli array dinamici sono una struttura che ha I seguenti membri:

- un puntatore (stesso tipo degli elementi dell'array) all'inizio dell'array;
- una lunghezza (tipo dipende dalla piattaforma e dall'implementazione).

STRINGHE

Le stringhe statiche sono degli array statici di byte (uint8) con protezione solo lettura.
Questo è il tipo che assumono I valori letterali di stringa.

Le stringhe dinamiche sono array dinamici di byte (uint8).

SCOPI

Il typechecker e il generatore di codice utilizzano lo stesso sistema di dichiarazioni e di scopi.
Ogni scopo è un codice univoco incrementale, che viene inserito nello stack degli scopi quando diventa attivo (ad esempio quando entriamo dentro un nuovo blocco di codice), e viene rimosso quando questo stato scade (quando usciamo dal blocco di codice).
Una dichiarazione è una chiave nella tabella delle dichiarazioni, ed è composta da un nome e uno scopo.
Il valore associato alle dichiarazioni è composto da

- un tipo (typechecker);
- un tipo e una regione dello stack della funzione (generatore di codice).

TYPECHECKER

Questa fase aggiunge i tipi di dato alle espressioni nell'AST, generando un AST tipizzato.
Definisce le regole di tipizzazione del linguaggio, ovvero: quali operazioni sono permesse su un tipo di dato, l'uguaglianza di due tipi, la compatibilità tra tipi diversi.
In questa analisi, l'AST viene attraversato dal basso verso l'alto, in modo tale da poter stabilire i tipi delle espressioni più complesse partendo da quelle più semplici.

UGUAGLIANZA

- Due tipi **primitivi**, **puntatori** o tipi di **array/stringhe statici** sono uguali se hanno lo stesso nome e la stessa dimensione.

- Due tipi **struttura** sono uguali se hanno lo stesso nome, la stessa dimensione, lo stesso numero di membri, e tutti i loro membri hanno lo stesso nome, lo stesso offset, e tipi uguali.

COMPATIBILITÃ

- Due tipi uguali sono sempre compatibili;
- Gli **interi letterali** sono compatibili con tutti gli altri tipi interi, e viceversa;
- I **puntatori generici** sono compatibili tutti gli altri puntatori, e viceversa;
- Gli **array generici** dinamici e statici sono compatibili con tutti gli altri array, e viceversa;
- Le **stringhe statiche** sono compatibili con le **stringhe dinamiche**.

OPERATORI

Binari

In ognuno di questi operatori si asserisce che i due tipi siano compatibili.

`+`

`+=`

stringa -- stringa ==> stringa
intero -- intero ==> intero

`*`

`*=`

intero -- stringa ==> stringa
stringa -- intero ==> stringa
intero -- intero ==> intero

`-, /, %, &, ^, |, <<, >>`

`-=, /=, %=, &=, ^=, |=, <<=, >>=`

intero -- intero ==> intero

`>, <, >=, <=`

intero -- intero ==> booleano

`==, !=`
qualsiasi -- qualsiasi ==> booleano

`&&, ||`
booleano -- booleano ==> booleano

`=`
qualsiasi di sinistra -- qualsiasi ==> qualsiasi

`.`
struttura/array/stringa -- qualsiasi ==> qualsiasi

Unitari

`~, -`
intero ==> intero

`++, --`
intero di sinistra ==> intero

`!`
booleano ==> booleano

`&`
qualsiasi di sinistra ==> puntatore

`*`
puntatore ==> qualsiasi di sinistra

`[x]`
array/stringa ==> qualsiasi

CORPI DI CODICE

Se il corpo ha già un tipo, è perchè al suo interno esiste un `return` (solo se è un corpo di funzione), o perchè il corpo ha un risultato.

Nel caso del corpo di una funzione:

- Il risultato e tutti i `return` devono avere tipi compatibili;
- Se esiste un risultato dentro il corpo, si può affermare che quest'ultimo restituisca sempre un valore.

Se esiste un figlio che restituisce sempre un valore, il corpo farà altrettanto.

IF

I risultati dei due corpi (vero e falso) devono essere compatibili.
Se I due corpi restituiscono sempre, anche l'if farà altrettanto.
L'if è un'espressione e il suo tipo è quello di uno dei due corpi.

SWITCH

I risultati di tutti i casi (case) devono essere compatibili.
Se i tutti i corpi restituiscono sempre, anche lo switch farà altrettanto.
Lo switch è un'espressione e il suo tipo è quello di uno dei corpi.

DEFINIZIONE DI FUNZIONE

Il tipo espresso nella dichiarazione di una funzione, deve essere compatibile al tipo effettivo del corpo di codice.

RETURN

Il return intuitivamente è un nodo che "restituisce sempre".
Il tipo del return deve essere compatibile con Il tipo espresso nella dichiarazione della funzione corrente.
Se il corpo della funzione corrente non ha ancora un tipo, esso sarà impostato al tipo del return; se il corpo ha già un tipo, esso deve essere compatibile con quello del return.

CASTING

I tipi “trasformabili” sono solo quelli interi, che possono essere trasformati da/a qualunque tipo intero.

DEFINIZIONE DI STRUTTURA

Ogni definizione di struttura risulta in un nuovo tipo nello scopo dei simboli attuale.

La lista dei membri della struttura viene riempita con le definizioni dentro il corpo della della struttura: vengono stabiliti gli offset dei membri che comprendono la somma delle dimensioni dei membri precedenti e degli eventuali “padding”.
La dimensione finale della struttura sarà la somma di tutte le dimensioni e i padding.

La dimensione esatta dei padding dipende dall’implementazione e dalla piattaforma.

BOUNDARY CHECKING

Quando indirizziamo un’array statico o una stringa statica, utilizzando un valore intero letterale, abbiamo la possibilità di controllare che l’indice rientri nella dimensione della struttura durante la fase di typechecking.

OPERAZIONI SU STRINGHE STATICHE

Le stringhe supportano due operazioni binarie:

- + (concatenazione): si giustappongono due stringhe;
- * (ripetizione) : si giustappone una stringa a se stessa *n* volte.

Quando entrambi gli operatori sono valori statici e/o letterali, possiamo portare a termine queste operazioni nel typechecker.

CODE GENERATOR

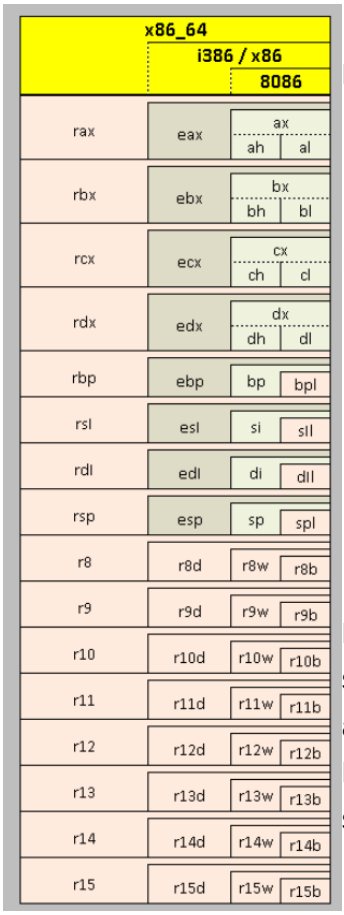
Una volta ottenuto l'albero sintattico completo, possiamo generare il codice in un linguaggio più semplice, che in questa implementazione è assembler x64 per GNU Linux. Alcuni compilatori (ad esempio TCC, "Tiny C Compiler"), piuttosto che generare codice assembler, generano direttamente un file binario eseguibile.

Questa analisi, come il typechecker, è un attraversamento ricorsivo dal basso verso l'alto. La funzione CodeGen() restituisce un output composto da:

- Codice (segmenti "text" e "data");
- Risultato (operando).

REGISTRI

Nella CPU I registri non sono illimitati, a differenza delle variabili del linguaggio.
Quando dobbiamo caricare una variabile in un registro, ci dobbiamo assicurare di salvare il contenuto del registro in memoria.
Dobbiamo quindi trovare un modo di allocare i registri che minimzzi il numero di letture /scritture della memoria.
Esistono diversi algoritmi di allocazione dei registri; uno dei più consolidati è il metodo che utilizza la colorazione dei grafi.
Tuttavia, questa implementazione non utilizza alcun algoritmo di allocazione: i registri sono allocati e svuotati senza alcuna regola di ottimizzazione.



Fondamentalmente, i registri sono suddivisibili in tre tipi:

- KIND_QDWBb
- KIND_QDWB
- KIND_QDW

In questo modo, possiamo rappresentare I registri come:

- una classe
- un indice di “sottoregistro”:
  - SUB_Q
  - SUB_D
  - SUB_W
  - SUB_B
  - SUB_b

Nei tipi KIND_QDW e KIND_QDWB, quando un sottoregistro viene allocato, vengono allocati anche tutti gli altri.

Nel tipo KIND_QDWBb, I sottoregistri SUB_b o SUB_B, sono indipendenti l’uno dall’altro.

L’utilizzo dei registri è regolato dalla convenzione delle chiamate System V AMD64 ABI:
Registri salvati dalla funzione chiamata: RBX, RSP, RBP, and R12–R15;
Tutti gli altri registri devono essere salvati dalla funzione che chiama, se intende preservarne i contenuti.

STACK

Lo stack utilizzato viene riservato all’inizio della funzione, e reimpostato alla fine. Secondo l’ABI, possiamo riservare solo multipli di 16 byte.

Una regione di stack è rappresentata da un offset da RBP (base dello “stack frame” della funzione), e una dimensione.

SFIDE, OPPORTUNITA' E SVILUPPI FUTURI

LINGUAGGIO

Si propone l'implementazione di costrutti di typechecking più complessi:

- Funzioni generiche statiche;
- Parametrizzare funzioni usando tipi;
- Aggiungere numeri reali.

COMPILATORE

- Controllo degli argomenti delle chiamate;
- Staccarsi dalla libreria C: aggiungere un'opzione per disattivare le librerie cstd;
- Individuare cicli di dipendenze nelle direttive ` @import `del pre-processore;
- Rendere il generatore di codice più indipendente dalla piattaforma;
- Scrivere un vero allocatore dei registri;
- Ottimizzazione.

LIBRERIE

Scrivere una libreria per sostituire la libreria standard di C.

BIBLIOGRAFIA - SITOGRAFIA

https://www3.nd.edu/~dthain/courses/cse40243/fall2019/

Jonathan Blow - https://www.youtube.com/watch?v=MnctEW1oL-E/

https://en.wikipedia.org/wiki/X86_calling_conventions

https://s3.amazonaws.com/media-p.slid.es/uploads/122159/images/1339091/x86_64-registers.png

https://www.felixcloutier.com/x86/index.html/

TCC - https://github.com/TinyCC/tinycc/ -- https://bellard.org/tcc/

https://gcc.gnu.org/pub/gcc/summit/2003/Graph%20Coloring%20Register%20Allocation.pdf/

https://gcc.gnu.org/wiki/

CONCLUSIONI

Senza alcuna conoscenza pregressa della materia, tutti i problemi riguardanti la progettazione del compilatore e del linguaggio sono stati affrontati con curiosità e spirito creativo. In conclusione, il percorso di apprendimento è stato molto istruttivo. Si sono acquisite conoscenze e competenze su diverse problematiche, ma si è riscontrato che il back-end risulta più complicato da gestire rispetto al front-end. Inoltre, si è constatato che l'allocazione dei registri rappresenta un'importante sfida che va affrontata con attenzione. Infine, la questione della sicurezza della memoria è estremamente complessa e richiede una soluzione accurata: sebbene il boundary checking degli array e delle stringhe sia un valido strumento, non è sufficiente per garantire una protezione completa. In generale, questo percorso di studio ha fornito importanti spunti di riflessione sulla complessità della programmazione e della gestione della memoria.