Faculteit industriële ingenieurswetenschappen Campus De Nayer
Elektronica-ICT fase 3
Academiejaar 2023-2024

Labo Datastructuren en algoritmen

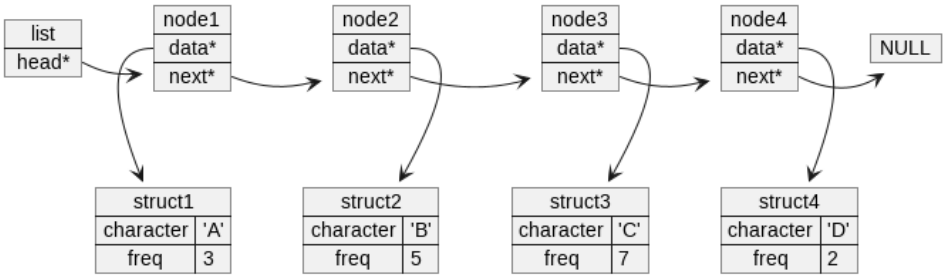
Ann Philips — Jeroen Van Aken

1 Gelinkte lijsten

Leerdoelen

- Het gebruik van 'stack' en 'heap' begrijpen en geheugengebruik van een programma in kaart kunnen brengen.
- Pointer-logica (referencing, de-referencing, functie-pointers) correct kunnen toepassen en geheugenlekken kunnen opsporen.
- Unit-test frameworks zinvol kunnen gebruiken.
- Compileren met behulp van een makefile.

Ooit in je leven zou je een single of double linked pointer list moeten hebben geprogrammeerd voordat je jezelf een software programmeur mag noemen, of, nog beter, een 'onderscheiden' C programmeur! Wel, dit is het moment om die horde te nemen ... De volgende oefeningen leiden je in een paar stappen naar een volledige implementatie van een enkelvoudig gekoppelde pointerlijst die elk elementtype aankan met behulp van callback-functies voor elementspecifieke bewerkingen. Figuur 1 geeft je een idee hoe deze lijst in het geheugen geïmplementeerd zal zijn. De implementatie van deze enkelvoudig gekoppelde pointerlijst zal worden gearchiveerd in een bibliotheek (zie toekomstige oefeningen) zodat deze kan worden hergebruikt in andere toepassingen.



Figuur 1: Enkel gelinkte lijst

Je hoeft niet vanaf nul te beginnen. Op Toledo vind je de sjabloonbestanden `splist_test.c`, `splist.c`, en `splist.h`. In `splist.h` vind je een basislijst van operatoren die je geacht wordt te implementeren. Het bestand `splist.c` bevat de echte typedefinities van de pointerlijst en een voorbeeldimplementatie van enkele van de lijstoperatoren. Dit is voorbeeldcode om je op weg te helpen - je bent vrij om het te veranderen, te verbeteren of zelfs volledig te vervangen door je eigen code.

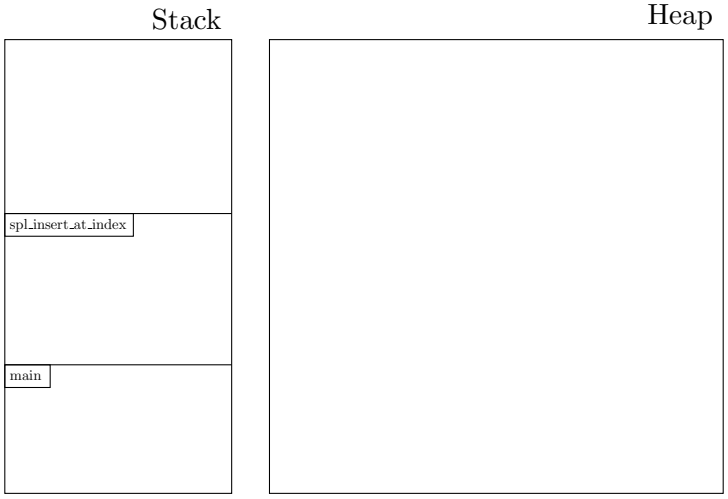
Als je eenmaal een lijstoperator hebt geïmplementeerd, start dan onmiddellijk met testen! Wacht niet te lang met testen omdat de complexiteit en de hoeveelheid fouten je te veel zou kunnen overweldigen. Zoals je inmiddels misschien wel geleerd hebt, testen echte software engineers code met behulp van unit-tests. Tijdens deze oefeningen zul je gebruik maken van het 'check' test framework voor C. De Makefile helpt je vervolgens om deze testen te compileren en uit te voeren. Kijk voor meer informatie over dit unit-test framework in de online documentatie op : `https://libcheck.github.io/check/`

Als alle operatoren geïmplementeerd en getest zijn, gebruik dan Valgrind om te controleren op geheugenlekken.

1.1 Analyse

Voordat je begint, moet je een duidelijk beeld hebben van wat je gaat implementeren. Om meer inzicht te krijgen in wat er gebeurt met alle verschillende pointers wanneer elementen worden toegevoegd of verwijderd uit de lijst, beginnen we met het schetsen van de geheugenlayout tijdens deze verschillende scenario's.

Je kan op Toledo een demonstratiefilmpje vinden over hoe je stack- en heap-tekeningen maakt voor een willekeurig stuk C-code. Bekijk vervolgens in het bestand `splist.c` de methode `spl_insert_at_reference()` en maak tekeningen (gewoon met pen en papier) voor alle 4 de gevallen die in commentaar worden aangegeven. Start voor elk van de gevallen met een tekening zoals in figuur 2



Figuur 2: Stack - Heap tekening

1.2 Implementatie

Voor de implementatie van je eigen lijst kan je beginnen met de code voor de methode `spl_create()`, `spl_insert_at_reference()` en `spl_size()`. Controleer bij elke stap de tekeningen die je tijdens de analysefase hebt gemaakt om een duidelijk beeld te krijgen van welke pointers in elk van de stappen moeten worden gewijzigd. Het helpt je ook met inzicht te krijgen wanneer je juist een pointer en een geheugenadres moet gebruiken (cfr. `int* a = &b;`) of wanneer je moet de-referencen en de werkelijke waarde van de variable moet gebruiken (cft. `int b = *a;`).

Vervolgens kan je starten met `spl_get_element_at_reference()` om de data van een bepaald element op te vragen. De methoden `spl_get_[first|last|next]_reference()` bevatten de implementatie om een bepaalde positie in de lijst op te vragen. Let telkens goed op alle randcondities die gegeven zijn in `splist.h`.

Verder kan je dan de `spl_remove_at_reference()` methode implementeren. Deze zal je helpen bij het implementeren van de `spl_free()` methode.

De argumenten van de laatste twee methoden `spl_get_reference_of_element()` en `spl_insert_sorted()` zien er op het eerste zicht een beetje vreemd uit. Dit komt omdat een van de parameters een callback functie is, die zal worden gebruikt om twee elementen uit de lijst te vergelijken. Op deze manier wordt de vergelijkingsmethode niet hard gecodeerd in de lijst en moet deze worden aangeleverd door de programmeur die je lijstimplementatie gebruikt. Als je een lijst gebruikt zoals in figuur 1, kun je kiezen om een index van een element te krijgen op basis van een letter, de frequentie of zelfs beide, alles hangt af van de implementatie van de vergelijkingsmethode.

In figuur 3 zie je een voorbeeld van het werkingsprincipe van functiepointers. De variabele callback, gedeclareerd als een functiepointer met een bepaalde signatuur, kan worden geladen met de methode `som()` of `prod()`. Zodra de functiepointer is toegewezen aan de callback-variabele, kun je deze gebruiken alsof het een gewone methode is. Merk op dat hoewel het lijkt dat op regel 15 en regel 17 dezelfde methode wordt aangeroepen, de waarde van variabele `resultaat1` en `resultaat2` verschillend zullen zijn.

```c
int sum(int x, int y) {
    return x + y;
}

int prod(int x, int y) {
    return x * y;
}

int main(int argc, char* argv) {
    int a, b;
    int (*callback)(int x, int y);
    a = 5; b = 7;

    callback = sum;
    int result1 = callback(a,b);
    callback = prod;
    int result2 = callback(a,b);

    return 0;
}
```

Figuur 3: Code voorbeeld met callback functie

1.3 Testen

Tijdens alle practica houden we ons aan het principe 'ongeteste code is nutteloze code'. Daarom zullen alle cases (zelfs edge-cases) voor onze lijst unit-getest worden. Hoewel in deze labtekst 'Testen' een aparte paragraaf is, zou het schrijven van tests gedaan moeten worden tijdens of zelfs voor je eigenlijke implementatie.

Het bestand `splist_test.c` bevat de main-methode bij het compileren en uitvoeren van je applicatie. In deze main-methode wordt het test-framework geïnitialiseerd en worden alle tests toegevoegd. De eigenlijke implementatie van de test staat bovenaan dit bestand. Binnen de tests wordt een klein stukje code uitgevoerd en gevolgd door een `ck_assert_msg()` methode, die het gedrag van de voorafgaande code evalueert op basis van een gegeven voorwaarde.

Er worden maar een paar testen gegeven, het is jouw verantwoordelijkheid om extra testen toe te voegen om ervoor te zorgen dat je code bugvrij is.

2 Boomstructuren

Leerdoelen

- Het werkingsprincipe van Huffmancodering begrijpen en kunnen toepassen.
- Statische en dynamische bibliotheken kunnen compileren en linken.
- 'code-coverage' van een testprogramma kunnen genereren en analyseren.

In deze oefening implementeren we een boomstructuur die zal helpen om een reeks data te comprimeren volgens een Huffmancodering. We starten met een beperkt aantal karakters met bijhorende frequentie zoals te zien in figuur 4a, maar de code kan uitbreidbaar zijn naar het hele alfabet of zelf willekeurige bytes.

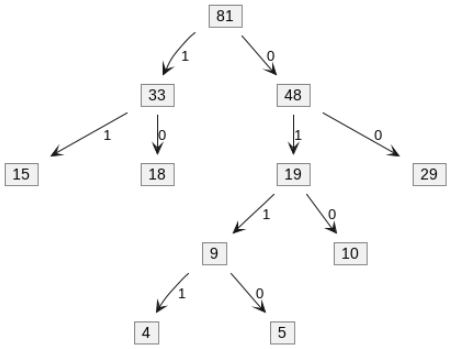
Startende van deze frequentietabel kunnen we een 'Huffmanboom' bouwen zoals te zien in figuur 4b. Hieruit halen we dan de bit code voor elk karakter in de frequentietabel. Merk op dat de lengte van deze code variabel is. Zes karakters zouden we eenvoudig kunnen coderen met 3 bits. De Huffman code voor elk van deze karakters varieert van 2 naar 4 bits, waarbij karakters die meer voorkomen een kortere codering krijgen.

Voor de implementatie van deze oefening gebruiken we de gelinkte lijst uit vorige opgave. Je kan deze als een statische library compileren en samen met je code linken. Of compileren als een dynamische library zodat deze at-runtime kan gebruikt worden.

Ook hier hoef je niet vanaf nul te beginnen. Op Toledo vind je een zip bestand met alle startcode. Hierin kan je `huftree.h` vinden met de beschrijving van de methoden die je zal moeten implementeren. De

letter	freq	bit code
r	4	0111
t	5	0110
u	10	010
a	15	11
d	18	10
e	29	00

(a) karakter-frequentie data



(b) Huffmanboom

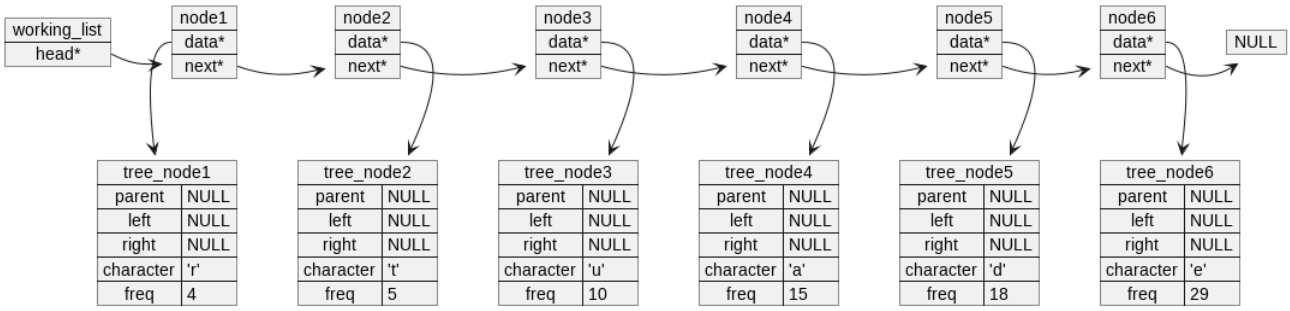
Figuur 4: Huffmancoderingsprincipe

`splist.h` file is gelijk aan deze van vorige oefening, enkel `typedef struct {...} element_t` krijgt hier een andere invulling.

2.1 Analyse

Ook nu beginnen we met een analyse van de oefening op papier. We maken terug Stack en Heap tekeningen voor de verschillende gevallen tijdens het opbouwen van de Huffmanboom.

We starten met een gesorteerde lijst op frequentie van alle karakters zoals te zien in figuur 5. Elke letter zit hier vervat in een datastructuur waaruit we de boom verder kunnen opbouwen. Voor elke `tree_node` kunnen we zowel de linker als de rechter kind-node linken. Ook is het handig alvast een link te voorzien vanuit elke node naar zijn bovenliggende 'parent'



Figuur 5: Gesorteerde karakter-frequentie lijst

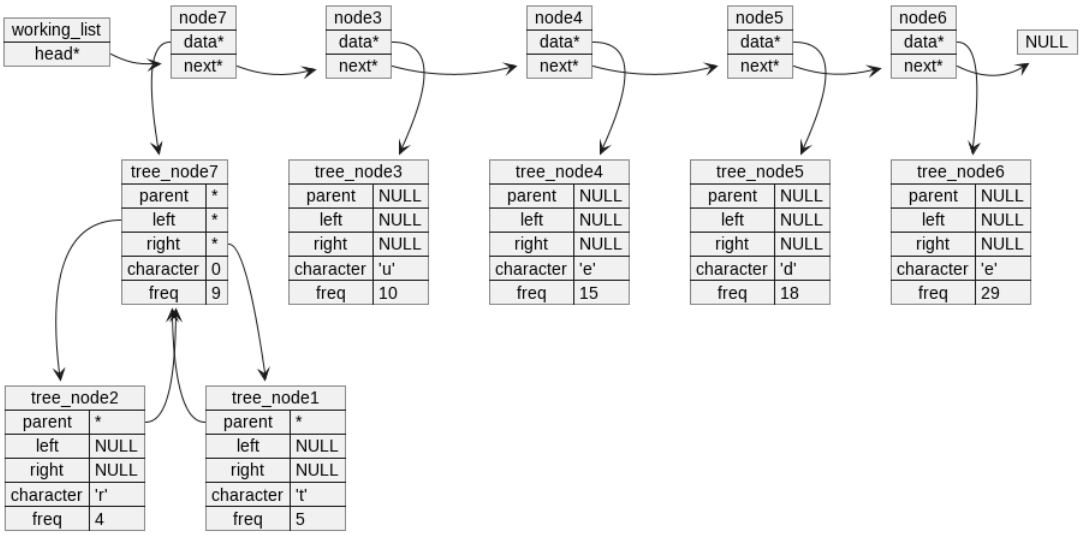
Startende van deze lijst verwijderen we de twee nodes met de laagste frequentie. We linken deze aan een nieuw aan te maken node, waarvan we de frequentie initialiseren op de som van zijn twee kind-nodes. We voegen deze node via `spl_insert_sorted()` dan weer toe aan de originele lijst zoals je kan zien in figuur 6. En herhalen deze stappen tot er maar één node meer in de lijst zit, dit zal dan onze 'root' node van de boomstructuur worden.

Maak nu zelf de schetsen voor alle volgende stappen, tot je de volledige boomstructuur vergelijkbaar met figuur 4b hebt kunnen opbouwen. Deze schetsen zullen je helpen bij de verdere implementatie.

2.2 Implementatie

In het `huftree.h` bestand heb je een overzicht van alle functies die je in deze oefening moet implementeren. Start hier met de `huf_create()` voor het bouwen van de Huffmanboom volgens de strategie die je geschetst hebt in de analysefase. Merk op dat je in `huftree.c` je eigen 'compare' functies zal moeten voorzien voor gebruik met de `spl_insert_sorted()` methode. Je kan hier zelfs verschillede methoden voorzien; bv. voor wanneer je moet sorteren op frequentie of wanneer je moet zoeken op letter.

Na het bouwen van de boom kan je er ook voor zorgen dat deze zonder enige memory-leak kan opgeruimd worden met `huf_free()`. Voor het opruimen van een boomstructuur is een recursieve implementatie



Figuur 6: Huffmanboom bij stap 1

startende vanaf de 'root-node' een mogelijke optie.

In `huftree.c` zie je verder ook nog drie hulpmethoden om bits te manipuleren of te testen in een *lange* voorgedefiniëerde buffer. De methoden zullen deze buffer benaderen als een groep bytes van type `uint8_t` waarbinnen ze elke bit afzonderlijk kunnnen manipuleren.

Na het bouwen van de boom kan je de `huf_encode()` en `huf_decode()` methodes implementeren. Let wel dat de Huffman bit code loopt van 'root-node' naar de 'letter-node'. Wanneer je de boom in de omgekeerde richting doorloopt (van 'letter' naar 'root') zal je de bits dus nog moeten omdraaien. Eventueel kan je na het bouwen van de boom deze éénmalig volledig doorlopen en de bitcode voor elke letter bijhouden in een afzonderlijke tabel, dit zou het codeer-proces mogelijks efficiënter maken. Voor het decoderen volg je de bitsequentie telkens van 'root-node' naar de uiteindelijke letter.

Maak zelf een makefile aan om je code te compileren en testen.

2.3 Testen

Ook hier weer is het testen een verhaal dat hand in hand gaat met je implementatie. `huftree_test.c` geeft je een start voor de implementatie van een aantal testen. Voeg zeker nog gevallen toe om je code zo goed mogelijk te testen.

3 Backtracking

Leerdoelen

- Het werkingsprincipe van backtracking begrijpen en kunnen toepassen.
- Recursieve methoden kunnen implementeren en debuggen.

Met backtracking kunnen we een oplossing voor een gegeven probleem op een efficiëntere manier brute-force berekenen. Elk pad in de berekening kan worden afgebroken zodra aan een criterium is voldaan, een volledige oplossing voor dat pad hoeft dan niet meer te worden berekend.

In deze opdracht simuleren we een optimalisatie voor het toewijzen van een scheidsrechter bij een voetbalcompetitie met n teams. Wanneer elke ploeg tijdens de competitie een heen- en terugwedstrijd speelt tegen elk ander team zijn er $2(n-1)$ speeldagen nodig. Om alle wedstrijden te plannen zullen er per speeldag $n/2$ matches georganiseerd worden. Bijvoorbeeld : in een competitie met 6 teams zijn er 10 speeldagen met telkens 3 wedstrijden.

Als optimalisatie zoeken we de opeenvolgende wedstrijden voor een scheidsrechter, zodat hij een minimale totale afstand moet afleggen om zich telkens naar elke thuisploeg te verplaatsen. Je mag er vanuit gaan dat de scheidsrechter telkens blijft overnachten na de match en de volgende dag naar de nieuwe locatie zal rijden. Er zijn verder nog een aantal randvoorwaarden voor een eerlijk verloop van de competitie :

- In twee opeenvolgende speeldagen moet een scheidsrechter met verschillende teams op het veld staan. Zowel voor de thuis- als uit-ploeg van de match, mag hij de de volgende dag geen wedstrijd leiden.
- Bij het einde van de competitie moet de scheidsrechter minstens één keer bij elke thuisploeg op bezoek zijn geweest.

	T1	T2	T3	T4	T5	T6
T1	0	445	365	529	305	221
T2	445	0	80	137	490	115
T3	365	80	0	180	420	157
T4	529	137	180	0	580	408
T5	305	490	420	580	0	610
T6	221	115	157	408	610	0

(a) Afstanden tussen teams in km

	game A	game B	game C
dag 0	T1 - T5	T4 - T3	T6 - T2
dag 1	T1 - T2	T3 - T5	T6 - T4
dag 2	T1 - T6	T3 - T4	T5 - T2
dag 3	T2 - T4	T3 - T1	T5 - T6
dag 4	T2 - T5	T4 - T1	T6 - T3
dag 5	T3 - T2	T4 - T5	T6 - T1
dag 6	T1 - T3	T4 - T2	T6 - T5
dag 7	T1 - T4	T2 - T6	T5 - T3
dag 8	T2 - T1	T3 - T6	T5 - T4
dag 9	T2 - T3	T4 - T6	T5 - T1

(b) Competitieplanning

Figuur 7: Voorbeeld competitiedata (6 teams)

Een voorbeeld van een competitieplanning kan je vinden in figuur 7b. De meest optimale planning voor een scheidsrechter, rekening houdend met de randvoorwaarden in deze competitie, is aangegeven met de rode vakjes in de tabel. Hierbij houden we rekening met de afstanden tussen de verschillende locaties van de velden voor elke ploeg. Een overzicht van deze onderlinge afstanden kan je vinden in figuur 7a. Als hij bij de start op dag 0, wanneer hij reeds aan het stadion van team 1 is, zijn kilometerteller op nul zet, zal hij tegen het einde van de competitie in totaal 3061 km gereden hebben.

3.1 Analyse

Voor je start aan de implementatie kies je of je voor een recursieve of niet-recursieve oplossing zal gaan. Bij deze laatste optie zal je zelf een stack moeten implementeren om de verschillende tussenstappen op te slaan. Bij de recursieve optie gebruik je de (eindige) functie-stack om doorheen de verschillende mogelijkheden te lopen.

Denk ook na welke datastructuren je nodig hebt om de verschillende tussenstappen en/of het meest optimale pad op te slaan. Zijn er ook specifieke randgevallen (bv. de eerste of laatste speeldag) waarbij je algoritme een andere of specifiekere beslissing moet nemen.

3.2 Implementatie

Om je algoritme te testen kan je starten van `demo6.txt` of `demo8.txt` die je kan vinden op Toledo. Hierin kan je telkens de nodige gegevens vinden om het backtracking algoritme te starten. Je herkent hier het aantal teams die deelnemen aan de competitie, een tweedimensionale array met de afstanden tussen de verschillende teams en een overzicht van de geplande wedstrijden voor elke speeldag.

Bij je implementatie start je met het inlezen van deze gegevens en ze telkens in een gepaste datastructuur op te slaan in het geheugen. Vervolgens start je het backtracking algoritme om alle mogelijk combinaties te berekenen. En houd je, telkens je een mogelijke oplossing gevonden hebt, het pad met de minimale afstand bij. Bij het beëindigen van het programma print je een overzicht van het gevonden pad en het minimaal aantal kilometers. Figuur 8 is een illustratie van hoe je deze info op het schrem zou kunnen brengen. Maak de nodige `.c` files aan en maak gebruik van een makefile voor het compileren en testen.

```
minimal distance : 3061
1-5  ->    0 | 0
6-4  ->  221 | 221
5-2  ->  610 | 831
3-1  ->  420 | 1251
2-5  ->   80 | 1331
6-1  ->  115 | 1446
4-2  ->  408 | 1854
5-3  ->  580 | 2434
2-1  ->  490 | 2924
4-6  ->  137 | 3061
```

Figuur 8: voorbeeld output voor backtracking programma

4 Symbolentabel met hashing

Leerdoelen

- Het werkingsprincipe van een symbolentabel begrijpen en kunnen toepassen.
- Complexere makefiles met o.a. automatische variabelen kunnen opstellen en aanpassen.

Tijdens deze en volgende oefening bouwen we stap voor stap een lexicale analysator en parser die logische bewerkingen kan uitvoeren op 'variabelen' of 'symbolen' opgeslagen in een tabel met open adressering. Deze symbolentabel is dan ook de eerste stap in dit proces.

De naam van een variabele bestaat uit een opeenvolging van maximaal 7 kleine letters (a-z). We voegen deze toe aan een tabel met beperkte grootte en berekenen de index van elk element via *open addressing with linear probing*. Als *hash functie* voor het bepalen van de open adressering refereren we als een *'basis26 functie'*.

4.1 Analyse

Het berekenen van de hash-functie gebeurt op basis van 3 letters uit de naam van variabele. Elke letter krijgt een bepaald gewicht in de basis26 functie. De eerste letter krijgt het gewicht 26^0, de tweede 26^1 en de derde 26^3. Zo berekenen we voor elke variabelenaam een (niet unieke) numerieke representatie. Door de middelste 3 letters van het woord te kiezen proberen we zo veel mogelijk differentiatie tussen de numerieke voorstellingen te verkrijgen. Bij woorden met minder dan 3 letters vullen we de gewichten verder aan met nul-waarden. Een rekenvoorbeeld voor een aantal woorden kan je in figuur 9 vinden.

variablenaam	sleutel	berekening
appel	a**ppe**l	$'e' \times 26^2 +' p' \times 26^1 +' p' \times 26^0$
peer	p**eer**	$'r' \times 26^2 +' e' \times 26^1 +' e' \times 26^0$
zo	**zo\0**	$0 \times 26^2 +' o' \times 26^1 +' z' \times 26^0$

Figuur 9: Rekenvoorbeeld voor basis26 hash functie

De symbolentabel is een vaste array met een gedefinieerde grootte `HTSIZE`, gedefinieerd in `hash.h` als 53. Bij aanvang van het programma worden alle elementen in deze tabel geïnitialiseerd; een lege variabelenaam duid op een nog lege plaats in de tabel. Na het berekenen van de hash voor een bepaald element gebruik je de deling methode met lineaire probing om tot de juiste index in de tabel te komen.

4.2 Implementatie

Voor deze opgave kan je de startbestanden downloaden van Toledo. `hash.h` geeft een overzicht van de te implementeren methoden voor het opstellen en onderhouden van de *hash-table*. Merk op dat in het bestand `hash.c` reeds enkele statische methoden zijn toegevoegd om je code meer structuur te geven. In `symbol.c` heb je een basis testprogramma zoals afgebeeld in figuur 10. Variabelenamen worden ingelezen via STDIN, check telkens of deze niet langer zijn dan 7 karakters en voeg ze toe aan de symbolentabel. De `ht_print()`

```c
int main(int argc, char *argv[]) {
    char buffer[1024];;

    ht_init();
    while (scanf("%s%*c", buffer) != EOF) {
        int index = ht_install(buffer);
        printf("[%2d] %s\n", index, buffer);
    }
    ht_print();
    return 0;
}
```

Figuur 10: Basisprogramma voor het inlezen van symboolnamen

methode drukt een overzicht van alle elementen samen met hun index op het scherm. (tip: een EOF typen op STDIN doe je met de toetscombinatie ctrl+d)

Gebruik een makefile voor compileren en linken van deze oefening. De voorbeeld makefile die je kan downloaden van Toledo maakt gebruikt van automatische variabelen, probeer vanaf nu bij het aanpassen of opstellen van een makefile ook het gebruik van deze variabelen aan te houden.

5 Lexicale analysator : LEX

Leerdoelen

- Het werkingsprincipe van een lexicale analysator begrijpen en kunnen toepassen.
- Reguliere expressies kunnen opstellen en interpreteren.
- Een input bestand voor de GNU Flex generator kunnen opstellen.

Na de symbolentabel start de implementatie van een lexicale analysator die stukjes tekst kan herkennen en ophalen voor verwerking in de volgende stap. Voor het implementeren van een lexicale analysator maken we gebruik van de de GNU Flex generator.

5.1 Analyse

De generator neemt als input een tekstbestand met vier secties, gescheiden door `%{` en `%}` of tussen `%%`. We noemen dit een `.l` of 'lex' bestand. Een voorbeeld voorbeeld van een leeg lex bestand kan je zien in figuur 11.

```
%{
 /* code block at begin of output file */
%}

 /* definitions section (declare names for complex regex's) */

%%
 /* rules section */
%%

 /* user code block at end of output file */
```

Figuur 11: Structuur van input bestand voor GNU Flex generator

Op basis van dit bestand zal de Flex generator een `.yy.c` bestand genereren dat verder te compileren is met een gewone C-compiler. De eerste en laatste sectie binnen dit lex bestand zijn voorzien om standaard C-code schrijven, de generator zal deze secties respectievelijk aan het begin en einde van de output-file toevoegen. Deze delen gebruik je doorgaans om bepaalde *define's* toe te voegen bovenaan de output-file of, onderaan, extra methoden (eventueel een main-methode) voor gebruik doorheen de rest van het bestand.

In het midden zijn er secties voor declaraties en vertalingsregels. Op basis hiervan zal de generator de

`yylex()` methode genereren. Via reguliere expressies in de declaratiesectie kunnen stukjes code gestart worden in de vertalingsregels. Elk stukje code in een vertalingsregel heeft toegang tot globale variabelen en moet een geheel getal als return-waarde geven. Dit is dan ook telkens de return-waarde van de `yylex()` methode. Figuur 12 geeft je een voorbeeld van een input-file die bepaalde e-mail-adressen zal inlezen en inladen in een globale `buffer[]` variabele. De input-file is ook voorzien van een eenvoudige main-methode aan het einde van de file om de code te testen.

```
 /* code block at begin of output file */
%{
    char buffer[1024];
    #define EMAIL   1
%}

 /* definitions section (declare names for complex regex's) */
email   [a-z]+\.[a-z]@[a-z]+\.be

%%
 /* rules section */
{email}     { sscanf(yytext, "%s", &buffer);
              return EMAIL;  // code returned by yylex()
            }
%%
 /* user code block at end of output file */

int main(int argc,char *argv[]) {
    int code;
    do {
        code = yylex();

    } while( code != 0 );
    return 0;
}
```

Figuur 12: Voorbeeld input file voor Flex generator

5.2 Implementatie

Bouw nu zelf een lexicale analysator die variabelenamen kan inlezen en deze zal inladen in de symbolentabel uit oefening 4. Je voorziet ook een tweede vertaalregel die bij het herkennen van een hexadecimaal getal, zoals bv `0x7A3F` (start altijd met `0x`) dit zal inlezen en de decimale waarde afdrukt op het scherm.

Op Toledo kan je het startbestand `hexlex.l` vinden, gebruik vervolgens de makefile om de Flex generator te starten en de output-file te compileren.

6 Token parser : YACC

Leerdoelen

- Het werkingsprincipe van een token parser begrijpen en kunnen toepassen.
- Een input bestand voor de GNU Bison generator kunnen opstellen.
- GNU Flex & GNU Bison kunnen genereren, compileren en linken met behulp van een makefile.

Als laatste implementeren we een verwerkingsmodule die de data, tokens genaamd, komende van de lexicale analysator verder kan verwerken. Voor de implementatie van een token parser maken we gebruik van de GNU Bison generator.

6.1 Analyse

De opbouw van het input-bestand, wat in de meeste gevallen te herkennen is aan de `.y` extensie, voor deze generator is analoog aan dit van de Flex generator zoals te zien in figuur 11. Enkel de declaraties en de vertalingsregels hebben hier een andere syntax.

De Bison generator zal de inhoud van de `yyparse()` methode genereren op basis van de vertalingsregels. Deze methode roept intern de `yylex()` methode, afkomstig van de Flex generator, op. Hierdoor zal in de meeste gevallen elk `.y` bestand een bijhorend `.l` bestand hebben.

In de declaratiesectie staan alle *tokens* gedefinieerd die de `yylex()` methode kan teruggeven. In figuur 13 zien we bijvoorbeeld op lijn 8 het EMAIL-token. De Bison generator zal naast het `.tab.c` bestand als output ook een `.h` bestand voorzien waar deze tokens een `#define` krijgen. We hoeven deze dus zelf niet meer te definiëren zoals in het voorbeeld van figuur 12, maar kunnen eenvoudig bijvoorbeeld `#include "myfirstyacc.tab.h"` toevoegen in het lex bestand.

```
1  %{
2      #define YYSTYPE int             /* type van yylval variable */
3      #include <stdio.h>
4      extern int yylex();             /* generated by flex in .l file */
5      extern int yyerror(char *);     /* called on error, can be overridden */
6  %}
7
8  %token  EMAIL
9
10 %%
11 input :                   { }   /* empty input */
12         | input command    { }
13 ;
14 command :  '\n'           { }   /* empty command */
15         | expr '\n'        { printf("-> %d\n", $1); }
16         | error '\n'       { yyerrok; }   /* command did not match any 'expr' */
17 ;
18 expr :   EMAIL             { $$ = $1; }   /* put yylval in $$ for further parsing */
19 ;
20 %%
21
22 int main(int argc,char *argv[]) {
23     yyparse();
24     return 0;
25 }
```

Figuur 13: Voorbeeld input bestand voor GNU Bison generator

De vertaalregels hebben elk een naam en kunnen via logische *grammar* of *produktie* aan elkaar worden gelinkt. Verder zijn ze opgebouwd uit één of meerdere semantische acties die starten na het verwerken van een token.

6.2 Implementatie

Op Toledo kan je ook nu weer de nodige startbestanden vinden. In de makefile kan je de target voor het uitvoeren van het bison commando vinden. Deze zal zowel de volledige implementatie in de `.tab.c` file genereren, als de `.tab.h` file voor gebruik in de `.l` file.

Schrijf nu een parser die, op basis van een lexicale analysator voor het vinden van hexadecimale getallen, binaire operaties op deze getallen kan uitvoeren. Elke input lijn (afgesloten met `'\n'`) is voor de parser een bit expressie; deze wordt geëvalueerd zoals aangegeven in figuur 14 en het resultaat wordt getoond op het scherm:

Orde		Operator	Voorbeeld	Resultaat
4	|	inclusieve of	0xaa83 | 0x3b71	0xbbf3
3	^	exclusieve of	0xaa83 ^ 0x3b71	0x91f2
2	&	en	0xaa83 & 0x3b71	0x2a01
1	~	inverteren	~0x3b71	0xffffc48e

Figuur 14: Overzicht binaire operaties voor GNU Bison parser

De binaire operatoren zijn rechts associatief, de unaire operator is niet associatief en de vorige opsomming geeft de prioriteit van laag naar hoog.

Combineer nu je code van opgave 4 en 5 tot een parser waarbij je hexadecimale waardes kan toekennen aan variabelen en binaire operatoren kan oproepen op deze variabelen. Wanneer je enkel een variabelenaam ingeeft drukt je de waarde van de variabele op het scherm. Een mogelijke output van je programma zie je in figuur 15

```
0xaa83 | 0x3b71
 -> 0xbbf3
appel = 0x33
peer = 0x66
banaan = appel | banaan
banaan
 -> 0x77
```

Figuur 15: Voorbeeldoutput voor interpreter van binaire operatoren