1 Introductie: doelen en opdrachten

Bij "Advanced Technical Programming" maken wij de hele cyclus van programmeren rond. Wij implementeren (en afhankelijk van je keuzes, ontwerpen wij zelfs) een programmeertaal. Dat wil zeggen, we gaan een *interpreter* en een basale *compiler* schrijven. Dit zal ons instaat stellen om in een eigen programmeertaal code te schrijven, die mee te compileren met C++ code, en die te draaien op een microprocessor.

Naast dat wij dit persoonlijk erg gaaf vinden om te doen, vergroot dit het begrip van programmeren door een overzicht te bieden van het hele proces van coderen tot runnen. Verder, zal dit de student de mogelijkheid geven om zelf talen te ontwerpen en te gebruiken, zelfs op microprocessoren. Nu zal men in de praktijk allicht geen volledige nieuwe compiler voor een volledige generieke programmeertaal schrijven, maar wat wel veel voorkomt in de praktijk zijn zogenaamde *domeinspecifieke programmeertalen*; dit zijn talen die ontwikkeld zijn om efficiënt, en vaak makkelijker, te kunnen coderen voor toepassingen die binnen een bedrijf, of applicatie, veel voorkomen. Zo hebben veel grote softwarepakketten een bijgeleverde scripttaal die gebruikt kan worden om snel macro's te maken en extensies te bouwen. Goede voorbeelden hiervan zijn bijvoorbeeld Unix Shell Script (jullie wel bekend), of bijvoorbeeld GameMaker Language (GML), dat gebruikt wordt om spellen snel te kunnen maken binnen GameMaker. Verder zou je ook een hoop markup languages zo kunnen beschouwen (bijvoorbeeld de markuptaal die op Wikipedia gebruikt wordt, MediaWiki). Wat een unieke skill is die je leert bij deze cursus – die niet veel mensen hebben – is een domeinspecifieke taal kunnen maken die je kunt runnen op een *microprocessor*!

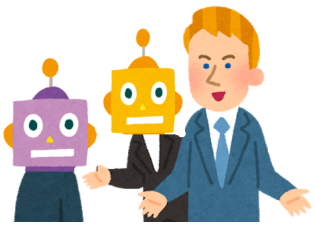
Een ander belangrijk doel dat wij hebben met de opdrachten, is het oefenen met de nieuwe programmeertechieken (in Python) die we leren bij de cursus. We zullen dus bij de opdrachten eisen stellen aan de stijl en technieken die gebruikt moeten worden bij het maken van de opdrachten.

Disclaimer voor jaar 3 NB: dit is een derdejaarsvak. Een van de doelen daarbij is om een volledig systeem van begin tot eind te kunnen bedenken, maken, en testen. Wij zullen dus weinig "voorzeggen", wij zullen dus vooral designprincipes en technieken aanreiken, maar gaan jullie daarna – expres – enigzins in het diepe gooien. Geen zorgen, jullie kunnen dit (jullie willen dit, jullie kunnen dit, tjakka tjakka tjakka!).

Opdracht 1: een interpreter De beoordeling van deze cursus hangt af van twee opdrachten. In de eerste gaan jullie een interpreter bouwen; een programma dat code in een andere programmeertaal on-the-fly can interpreteren en uitvoeren. Dit gebeurt dus zonder het te vertalen naar machine-code. Het is belangrijk voor het maken van de eerste opdracht de tweede opdracht alvast goed te lezen, namelijk de datatypes en code wordt grotendeels hergebruikt voor de tweede opdracht.

Opdracht 2: een compiler Interpreters zijn vaak prima voor PC's. Python zelf is bijvoorbeeld een interpreted taal. Echter, voor een microprocessor willen we natuurlijk niet een interpreter hebben draaien die vervolgens andere code interpreteert. Daarom gaan we een basale compiler bouwen, die één functie aan code kan omzetten in Assembly (heel dicht op machineinstructies). Daarmee gaan we een functie implementeren die we gaan meecompileren met C++ -code (met een Make-file), en draaien op een microprocessor (de Arduino Due).

Beoordeling Bij de opdrachten zullen wij op een aantal dingen letten. Ten eerste, zitten bij beide opdrachten een stel must-haves; dit zijn eisen aan de code en functionaliteiten die goed moeten zijn wil je een voldoende kunnen halen. Voldoen aan de must-haves levert dus een 5,5 op. Voor alle overige punten kun je extra functionaliteiten implementeren die in de Should/Could-haves staan.
Je mag ook zelf functionaliteiten voorstellen, maar die moeten dan wel even worden afgestemd met de docent om het maximaal haalbare aantal punten vast te stellen. Wij waarderen doorzettingsvermogen en creativiteit.[1]



Figuur 1: Het "tolken" (interpreting) van code voor de machine.

2 Interpreters en Compilers

Een interpreter is een programma dat code on-the-fly kan uitvoeren, zonder dit naar machinecode te vertalen. Dit heeft uiteraard het grote voordeel dat de code uitgevoerd kan worden op elk systeem waarvoor er een interpreter beschikbaar is. Geïnterpreteerde talen zijn er legio. Python is waarschijnlijk een van de bekendste voorbeelden van geïnterpreteerde talen.

Wat een interpreter moet doen, is

1. de code analyseren in termen van uit welke keywords, variabelen, etc. deze bestaat (Lexen),
2. de resulterende lijst van tokens omzetten in een uitvoerbare datastructuur – een zogenaamde abstract syntax tree (AST, zie paragraaf 2.2)[2] – en die in het geheugen zetten (parsen), en tenslotte
3. het programma (d.w.z. de AST) uitvoeren, binnen zijn eigen programma.

Het is deze laatste stap die een interpreter onderscheid van een compiler. Een compiler moet in de vierde stap de AST niet uitvoeren, maar vertalen naar machine-code. Oftewel, vertalen naar een executable die dan we door een operating system, dan wel direct op een processor uitgevoerd kan worden.

Wij moeten hierbij aantekenen dat onze compiler niet helemaal naar machinecode zal gaan, maar naar één abstractielaagje daarboven, namelijk Assembler. Assembler is kort-gezegd alleen maar een leesbare versie van machinecode. Dit heeft voor ons dus als

[1] Vorig jaar hebben wij meerdere tienen kunnen uitdelen, en zelfs honourssterren aangevraagd voor bijzonder gave uitwerkingen waar veel extra's inzat.

[2] Hoewel een AST de datastructuur is die in de literatuur het meest gebruikt wordt is, is het niet altijd noodzakelijk, of zelfs mogelijk, om dit te gebruiken. Bijvoorbeeld, Jens Bouman (die ATP deed in 2019-2020) heeft een `Piet`-interpreter geschreven. Omdat Piet een taal is die niet met regels code maar een 2D canvas werkt, is een boom niet de juiste datastructuur. In plaats daarvan werkt Jens' `Piet`-interpreter met een *abstract syntax graph*. Als je geïnteresserd bent hoe dit werkt, je kan het project vinden op: `https://github.com/JensBouman/Piet_interpreter`.

voordeel dat we a) de output van onze compiler nog zelf kunnen lezen, en b) dat we het resultaat daarna nog weer mee kunnen compileren binnen een groter geheel, zoals een C++ -project.

Omdat alleen de laatste stap van interpreteren en compileren verschilt, zullen wij de eerste stappen (tokenisen en parsen) samen behandelen. Daarna gaan we apart in op runnen in een interpreter, en compileren in een compiler.

2.1 Stap 1: Lexen

Bij de eerste stap, het *lexen* (uit het latijn *lex*), zetten we de platte tekst waaruit de code bestaat om in betekenisvolle losse brokjes – tokens – die we straks makkelijker kunnen interpreteren. Tokenisen ben je misschien gewend uit C++ en andere programmeertalen als het opdelen van een `string` in woorden. Woorden zijn namelijk de standaard betekenisdragers in taal. Om een programmeertaal te analyseren, willen we echter graag iets meer weten, en iets meer detail aanbrengen in *wat voor* woorden en andere betekenisvolle elementen de tekst eigenlijk bevat. Laten we eens naar het volgende stukje code in C++ kijken, en ons afvragen uit welke betekenisvolle elementen deze code eigenlijk bestaat.

```
int fib(int n) {
    if(n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

Vanaf het begin zien wij:

- een primitief type: `int`
- een identifier: "`fib`"
- een haakje openen
- een primitief type: `int`
- een identifier: "`n`"
- een haakje sluiten
- een accolade openen
- een keyword: `if`
- een haakje openen
- een identifier: "`n`"
- een operator: `<=`
- een haakje sluiten
- etc.

Het maken van zo'n lijst van betekenisvolle losse elementen is het hele process van *lexen*, en het eerste van de drie stappen van een interpreter (lex-parse-run), en dan the compiler (lex-parse-build). Wat een betekenisvol element is om te interpreteren hangt heel erg af van de programmeertaal. Bijvoorbeeld, wij hebben in de bovenstaande lijst expliciet géén white-space of regelovergangen meegenomen, omdat het in C++ geschreven is. Als we deze functie echter in Python hadden geschreven, dan was de hoeveelheid inspringen na

regelovergangen wel relevant geweest, en hadden we bij elk element van deze lijst ook de *indentation* (en allicht het regelnummer) op moeten nemen.

Dit kan uiteraard allemaal nog een stuk extremer: bijvoorbeeld, in de esoterische programmeertaal *Shakespeare*[3] bijvoorbeeld heeft veel "artistic fluff". Bijvoorbeeld, het volgende stukje doet een pop vanaf een stack (elk "character" in Shakespeare heeft haar of zijn eigen stack):

Lady Macbeth:
Recall your imminent death!

De grap is dat het enige relevante stukje informatie op de tweede regel het keyword `recall` is, de rest staat er alleen maar om er een mooi verhaal van de maken. Dus, om het programma succesvol te lexen hoeft een interpreter alleen het keyword `recall` te onthouden.

De lijst van betekenisvolle elementen slaan we meestal op als een lijst van speciale objecten, zogenaamde *Tokens*. Hoe je een Token zou moeten definiëren is programmeertaalafhankelijk. Bovendien zou kan het ook afhangen van de functionaliteit die je interpreter of compiler later wil bieden. Bijvoorbeeld, voor het genereren van duidelijke foutmeldingen zou het handig kunnen zijn om de regelnummers en karakternummers op te slaan bij het Token, hoewel dat misschien gezien de taal op zich niet uitmaakt (zoals in C++).

Zoals er al gehint werd in de bovenstaande lijst, wordt meestal onderscheid gemaakt tussen verschillende types van Tokens. Dit zal dus meestal leiden tot een structuur van verschillende Token-classes met overerving.

2.2 Stap 2: Parsen

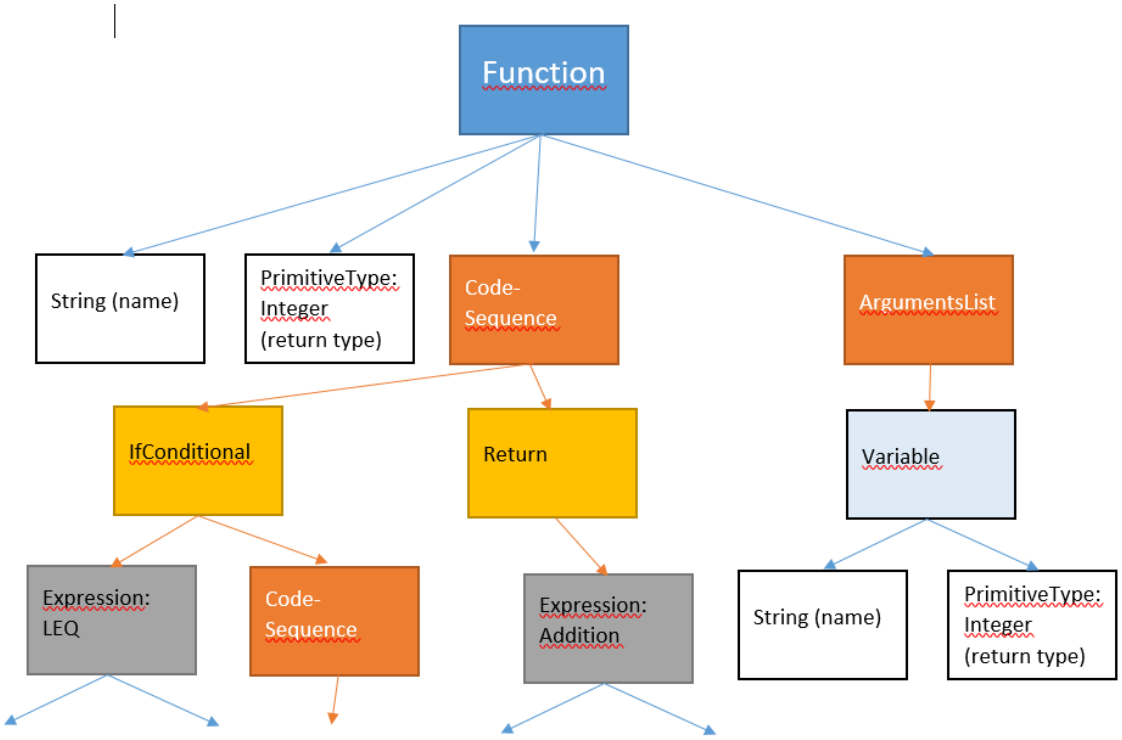
Nu we een lijst van tokens hebben, kunnen we het nog niet direct runnen als een programma. Met name, het moet omgezet worden in een structuur waar een interpreter (of compiler) makkelijk doorheen kan lopen om de flow van het programma te volgen. De makkelijkste structuur om dit te doen is een recursieve structuur. Zo'n recursieve structuur wordt meestal een *abstract syntax tree* genoemd.

Als we eens kijken naar het code voorbeeld van de vorige paragraaf, de Fibonacci-reeks, zouden we zoiets kunnen krijgen:

```
int fib(int n) {
    if(n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

Bovenaan de AST zien we dat dit stukje code een Function betreft. We hebben dus iets van een Function-object nodig, met als variabelen: een naam ("fib"), een returntype (in dit geval bijvoorbeeld een PrimitiveType int), een lijst van argumenten (voor dit geval bijvoorbeeld iets als ArgumentList [(Variable "n" (PrimitiveType int))]), en een CodeSequence (met de inhoud van functie). De CodeSequence zal een lijst met opeenvolgende statements moeten bevatten, in dit geval 2: een If met een Condition (een BooleanExpression bestaande uit een LEQ tussen de Variable n en de Constant 1) en weer een CodeSequence (waar in dit geval alleen een return-statement instaat), etc. Als we dit visueel weergeven in Figuur 2, kunnen we zien waarom dit een tree heet. Met name CodeSequences kunnen een enorme branching factor opleveren (het is immers een lange lijst van opeenvolgende statements). Het is ook duidelijk waarom dit een syntax-boom is. In de boom staan namelijk Objecten die overeenkomen met de syntax van de programmeertaal.

[3] `http://shakespearelang.sourceforge.net/report/shakespeare/`



Figuur 2: Het bovenste stuk van de AST

2.3 Stap 3 optie a: Runnen

Een AST is een structuur die makkelijk te interpreteren is voor een computer. Met een interpreter hoeven we eigenlijk aan de AST maar weinig toe te voegen om het programma te kunnen draaien.

Een interpreter zal bovenaan de AST (van het hele programma) beginnen. Dit kan bijvoorbeeld de main-function zijn, in het geval van C++ . Voordat dit echter kan gebeuren moet eerst de Program State van het programma worden geïnitialiseerd. Meestal betekent dit voor een interpreter om een aantal objecten aan te maken die het geheugen benodigd voor het programma representeren, en eventueel een outputstream, etc. Bij C++ moeten vervolgens ook de runtime-arguments (de argumenten van de mainfunction) in dit geheugen worden geplaatst. Daarna begint het runnen met het *uitvoeren* van de Root-node van de AST. Dit uitvoeren zal dus een van de belangrijkste functies van de interpreter zijn.

Laten we even uitgaan van de AST van Figuur 2, en het bijbehorende stukje code. Wat gebeurt er als deze functie wordt aangeroepen? De aanroep zal gebeuren vanuit een of andere FunctionCall elders in de AST. Uiteraard zal de interpreter ervoor moeten zorgen dat gecheckt wordt of de function call correct gebeurt. Dat wil zeggen, bestaat deze function, en zo ja, kloppen het aantal argumenten en het type? Vervolgens, omdat dit een function is, moet er een scope worden aangemaakt – een soort lokale state, met variabelen met namen die alleen voor deze function geldig zijn. (Eventueel kan de functie ook bij variabelen uit de globale scope, maar laten we dat nu even buiten beschouwing laten). In deze scope moeten de variabelen uit de arguments-list worden gezet, met de waarden die ze meekregen uit de function call. Oftewel, als de aanroep van de functie `fib(5)` was, komt er $n : 5$ in de scope te staan. Vervolgens wordt statement voor statement, de CodeSequence uitgevoerd (yes, dat zou prima weer die zelfde voer-uit-functie kunnen zijn). Eerst wordt de IfConditional uitgevoerd, met als eerste stap het evalueren van de LEQ-expressie. In de LEQ-expressie staat een variabele n - die staat in de scope, dus daar kunnen we nu 5 voor invullen, en een constante 1, dus daar hebben we een 1. Daarmee kunnen we de LEQ-

expressie evalueren naar een Boolean-resultaat `false`. Met dat resultaat kan het uitvoeren van de IfConditional door, namelijk, moeten we de CodeSequence gaan uitvoeren? In dit geval niet, omdat de conditie niet waar was, en dus is het uitvoren van de IfConditional klaar. Dus we gaan verder met het uitvoeren van de CodeSequence daarboven, en het volgende statement is een Return. Het uitvoeren van die return vereist weer het evalueren van een expressie, in dit geval een som. Maar in deze som staan nu FunctionCalls die we eerst moeten voltooien, etc. etc.

Oftewel, het runnen van de code met een AST is één groot recursief process, waarin statements worden uitgevoerd en expressies geëvalueerd, lokale scopes worden aangemaakt (en weer vernietigd na het voltooien van een functie), totdat de interpreter aan een einde van de AST is gekomen en het programma termineert. Dat klinkt eenvoudig, maar afhankelijk van hoe complex je programmeertaal is, kan dit uiteraard nog knap lastig worden.

2.4 Stap 3 optie b: Vertalen naar Assembler

In plaats van een programma dat de AST traverset en zelf een program state bijhoudt en aanpast (een interpreter), zouden we graag de program state direct op een computer kunnen zetten. Oftewel, voor het geheugen van het programma geen objecten aanmaken die vervolgens weer door een ander programma op een bijvoorbeeld een microcontroller in het geheugen moeten worden gezet. Maar om deze tolk – the middleman – uit het process te verwijderen moeten we rekening gaan houden met de beperkte set van mogelijke instructies op een controller, en de specifieke geheugenstructuur van de microcontroller. Oftewel, we gaan nu een compiler bouwen, die de code in de input-programmeertaal omzet naar machinecode (of in ons geval stiekem één laagje daarboven: Cortex-M0 Assembler code). Dit is uiteraard een lastiger probleem dan het maken van een interpreter – we moeten nu immers zelf het geheugenmanagement gaan afhandelen.

Als we al een interpreter hebben, kunnen veel daarvan hergebruiken om de compiler te bouwen. Namelijk, de lexer en de parser zijn hetzelfde. Ook het idee van het opbouwen van scopes gaat heel dichtbij het geheugenmanagement zitten wat we moeten doen op een microcontroller.

Register allocation Er zijn echter wel een paar hele grote verschillen. Ten eerste, moeten we exact gaan kiezen welk van het beschikbare (beperkte) geheugen, we daadwerkelijk gaan gebruiken. Als we dat efficiënt willen doen moeten we daar belangrijke keuzes in gaan, want niet elk soort geheugen is gelijk. Uitgaand van de Cortex-M0, hebben we verschillende stukjes geheugen:

- Registers, en dan met name:
  - de scratch registers R0-R3, die het goedkoopst zijn in gebruik, omdat we hun waardes niet veilig hoeven te stellen, en we er alles mee kunnen;
  - de overige lage registers R4-R7, waarmee we wel alle operaties kunnen uitvoeren, maar de waardes binnen functies wel veilig moeten stellen;
  - de hogere general-purpose registers R8-R12, die we wel kunnen gebruiken, maar waar we lang niet alle operaties op mogen uitvoeren;
  - de speciale registers, zoals de stack pointer en de program counter, die gereserveerd zijn en waar we dus alleen hele specifieke dingen mee mogen doen,
  - het data-segment;
  - het BSS-segment;
  - het code-segment;
  - en natuurlijk de Stack.

We gaan ons in deze cursus beperken tot een paar functies, die bovendien geen side-effects mogen hebben. Dit zal de practicumopdracht een stuk makkelijker maken, omdat we dan eigenlijk alleen rekening hoeven te houden met lokale scopes. Echter, je kan je voorstellen dat hoe groter een programma wordt, het bepalen van welke dingen op welk moment tegelijkertijd in het geheugen moeten staan, en welke registers dus op welk moment welke variabele moeten voorstellen een zeer lastig probleem is. Hoe meer we nodig hebben, hoe lastiger het gaat worden. We willen dus zo min mogelijk registers hoeven inzetten. Helaas is dit equivalent aan het *graph colouring* probleem, u wel bekend van ALDS, wat een NP-hard probleem is. We kunnen dit probleem dus vaak niet exact oplossen, en zullen moeten volstaan met een heuristieke aanpak. Compilers zoals GCC hebben een goed doordachte aanpak. Dat neemt niet weg dat het altijd beter kan, maar lang niet alle pogingen tot verbetering zijn hier succesvol in geweest. Voor meer informatie, zie bijvoorbeeld: `https://gcc.gnu.org/wiki/RegisterAllocation`. Een van de dingen waarvoor je punten zal kunnen krijgen in de opdracht is dus ook *slimme trucs* voor registerallocatie.

Enfin, hoe pakken we nu zoiets aan? Laten we een aantal observaties maken aan de hand van ons inmiddels welbekende code-voorbeeld in C++ :

```
int fib(int n) {
    if(n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

Als we de scope willen opbouwen voor deze functie, en daar registers voor willen alloceren vallen een aantal dingen op. Ten eerste, het register waarin de input binnenkomt staat vast: aan het begin van de functie heeft $R0$ de waarde van n. Echter, we zien ook direct dat $R0$ niet n kan blijven: er is een return-waarde, dus daar moet iets anders kunnen komen staan. Met name de regel:

```
return fib(n-1) + fib(n-2);
```

zou wel eens problematisch kunnen zijn: we hebben n nodig om de argumenten voor de function calls te berekenen, $R0$ om de resultaten van de function calls op te vangen, en bovendien moet aan het einde van het evalueren van de expressie $R0$ de waarde van de uitkomst krijgen, want dat is de waarde die wordt gereturnt. Oftewel, hoewel er maar één variabele is in deze functie, moeten er zeker meerdere registers worden gebruikt. Hoeveel we er maximaal nodig zullen hebben kunnen we opmaken uit de AST: een function call zou minstens 1 extra register nodig kunnen hebben (vanwege het gebruik van de returnwaarde) en maximaal het aantal argumenten dat meegegeven wordt (indien de registers om de argumenten in te zetten in een ander register veilig gesteld zouden moeten kunnen worden), een expressie als $n - 2$ zou een extra register nodig kunnen hebben, etc. etc.

De truc is natuurlijk wel dat niet al die dingen tegelijkeríjd nodig zijn. Bijvoorbeeld: $4 + 3 + 5 + 9$ zijn wel drie berekeningen, maar hebben maar één accumulator nodig om de deelresultaten in te verzamelen. Het herkennen van dat soort patronen, en daar een goede truc voor verzinnen om dat efficiënt te doen is de eerste stap in registerallocatie. Natuurlijk zal in eerste instantie een compiler wel altijd een programma moeten maken dat werkt. Pas in tweede instantie kunnen we op efficiënter gebruik van de registers gaan optimaliseren.

Stapje voor stapje Nadenken over geheugengebruik is natuurlijk niet het enige wat we moeten doen. Zoals gezegd zullen wij ons beperken tot enkele losse functies, zonder side-effects. Dat heeft een aantal belangrijke voordelen:

- We weten precies wat er binnenkomt, namelijk de argumenten van de functie;
- en we weten precies wat er uit moet komen, namelijk het return statement.

Oftewel, aan het begin hebben R0, R1, ... de waarde van de input parameters, en staat het LR op waar het programma weer doormoet als we returnen. Aan het eind moeten we R0 hebben staan op wat er gereturnt moet worden, en moeten we de waarde dat aan het begin op LR stond in PC zien te krijgen. Alles wat daartussen zit om deze input naar de output te krijgen zijn onze eigen hulpmiddelen, en kunnen we dus weer vergeten.[4] Laten we ons welbekende `fib`-voorbeeldje er weer bijpakken, beginnend bij de AST van Figuur 2.

Bovenaan de AST staat onze functie-definitie. Dat is mooi, want dat is precies wat we bij de opdracht zullen pakken. De lijst van argumenten (een van de child nodes van de Function) vertelt ons wat de argumenten zijn. Dit hoeven we zelf niet te programmeren dus, maar we moeten het wel weten. De functienaam is ook belangrijk om te weten; dit moet namelijk ook het label worden van de subroutine in onze `.asm`-file. Oftewel – ná de benodigde statements om aan te geven dat we cortex-m0 instructies gebruiken, in het `.text`-segment zitten etc. – zal onze file een `.global fib` moeten gaan bevatten, en het begin van de functie zal gelabeld zijn met `fib:`. Iets anders veiligs om te doen is even `LR` naar de stack push, om die later met iets als `pop {PC}` er weer af te kunnen halen.[5] Ook moet aan het begin van de functie natuurlijk alle gebruikte beschermde registers naar de stack gepusht worden, maar dat heeft weer te maken met register allocation.

Na het aanmaken van de functie kunnen we doorgaan naar de CodeSequence van de functie. Hier zien wij als eerste child-node een IfConditional (zonder else-clausule). Dit gaat dus betekenen dat we de Expression moeten gaan uitrekenen, en het resultaat daarvan gebruiken om al dan niet te branchen naar of de code sequence die onder de `if` zelf hangt, of het volgende statement na de `if`. Wij zien dat de expression een less-than-or-equal-to is. Hierbij kunnen we gaan kiezen hoe we dit oplossen. Persoonlijk ben ik er wel voor om te branchen op het tegenovergestelde (dus `BGT`) naar het volgende statement na de `if`, maar dat kan ook anders. Het voordeel daarvan is dat we onder dat `BGT`-statement direct kunnen doorgaan met de CodeSequence die onder de IfConditional hangt. Dus, wat moeten we boven de `BGT` uitrekenen om de comparison goed te krijgen? Als we naar de Expression:LEQ node in de AST kijken heeft die een left-hand side, namelijk de variabele n, die op R0 staat. In de right-handside staat een immediate, dus die kunnen we direct met `CMP R0,#1` kunnen vergelijken, en daarna met een `BGT label` naar óver de `if` heen kunnen springen. Als het wel kleiner of gelijk aan 1 blijkt te zijn kunnen komen we aan bij de `return`. Omdat n nogsteeds op `R0` staat, kunnen we nu het link-register (en de eventuele andere veilig gestelde variabelen) direct weer poppen.

Het statement ná de `if` is weer een return statement. Hier blijkt dat we een addition expression moeten doen, met twee function calls. Dit is uiteraard een stuk lastiger. Namelijk, we moeten nu een stapeltje variabelen gaan reserveren! Dit komt omdat we de `fib`-subroutine opnieuw willen aanroepen, en dus n, en het resultaat van de function calls, niet in scratch-variabelen kunnen zetten. Hoe `gcc` (met de optie `-O1`) dit doet is zo:

```
fib(int):
        push {r4, r5, r6, lr}
        mov r4, r0
        cmp r0, #1
        ble .L1
        sub r0, r4, #1
        bl fib(int)
        mov r5, r0
```

[4] Als vingeroefening, probeer eens na te gaan wat er precies anders is als we ook globale of object-variabelen mogen gebruiken in een functie.

[5] Aan het einde `MOV PC,LR` zou uiteraard ook kunnen, mits er geen andere subroutines worden aangeroepen in de functie.

```
        sub r0, r4, #2
        bl fib(int)
        add r0, r5, r0
.L1:
        pop {r4, r5, r6, lr}
        bx lr
```

Om verschillende opties om stapje voor stapje de AST om te zetten in Assemblercode, zou je kunnen kijken naar hoe de `gcc`-compiler dit doet voor C++ . Uiteraard kan je op de commandline zelf `.lss` en `.lst` files laten maken, maar makkelijker is het om te kijken naar `https://godbolt.org/`, onder met de `ARM gcc 9.2.1 (none)` compiler, en dan extra commandline opties mee te geven. Optimalisaties beginnen bij de optie `-O0` geen optimalisaties. `-O1` vind ik zelf meestal het duidelijkste, en hogere optimalisaties duurt vaak best wel even om door te krijgen qua shenanigans-niveau. Met `-O3` voor `fib` doet hij bijvoorbeeld dit:

```
fib(int):
        cmp r0, #1
        bxle lr
        push {r4, r5, r6, r7, r8, lr}
        mov r5, #0
        sub r7, r0, #2
        bic r3, r7, #1
        sub r6, r0, #3
        sub r6, r6, r3
        sub r4, r0, #1
.L3:
        mov r0, r4
        bl fib(int)
        sub r4, r4, #2
        cmp r4, r6
        add r5, r5, r0
        bne .L3
        and r0, r7, #1
        add r0, r0, r5
        pop {r4, r5, r6, r7, r8, lr}
        bx lr
```

Een andere optimalisatiesetting die interessant is om te bekijken is `-Os`, die op de size (hoeveelheid code) optimaliseert. Voor meer inspiratie; bekijk het filmpje op Canvas.

3 Opdracht 1: de interpreter

Als eerste opdracht gaan we een interpreter bouwen. De basisfunctionaliteit die we willen hebben is[6]: we implementeren één of meerdere functies in een zelf-gekozen of zelf-ontworpen programmeertaal. De interpreter moet dus minimaal:

- één functie per file kunnen supporten (mag ook meer)
- de interpreter moet de inputs van de functie kunnen meegeven (mag op de commandline, maar mag ook met een GUI)

[6] NB: dit is net iets anders dan de basisfunctionaliteit van vorig jaar

- een functie moet een andere functie kunnen callen (die in een andere file mag staan die de interpreter dan ook moet kunnen inladen)
- de interpreter moet het resultaat van de functie kunnen weergeven[7]

De eerste stap is dus het kiezen, of *ontwerpen* van de programmeertaal. Er zijn drie opties:

1. Een subset van een bestaande volledige programmeertaal. Echter, niet toegestaan zijn: C++ (die we als voorbeeldtaal gebruiken), Python (waarin de interpreter en compiler geschreven moeten worden), Brainfuck (want daar gebruiken we een voorbeeldtutorial van, en kan gebruikt worden voor het aantonen van Turing-compleetheid), en Assembler (wat juist de output moet zijn van de compiler). De subsets van bestaande programmeertalen dienen Turing-compleet te zijn. Voorbeelden uit deze categorie zijn bijvoorbeeld: COBOL[8], Fortran, Lisp, MATLAB, Ruby, maar ook bijvoorbeeld TeX (ja, dat is Turing-compleet).
2. Een esoterische programmeertaal; dit zijn programmeertalen die ontworpen zijn om bijvoorbeeld minimalistisch te zijn, of een programmeerconcept tot in het extreme te laten zien, of als lol- of kunstproject. Wederom, deze talen moeten Turing-compleet zijn. Veel van deze talen zijn te vinden op de Esolang-wiki: `https://esolangs.org/wiki/Category:Turing_complete`.
3. Een zelf-ontworpen Turing-complete programmeertaal.

NB: géén twee mensen mogen dezelfde taal kiezen. Wees er dus vroeg bij als je een bestaande taal wilt kiezen, want wie het eerst komt, wie het eerst maalt.

We zullen nu in detail de must-haves in detail bespreken. Let goed op dat je alle must-haves vervult. Indien de must-haves goed zijn volbracht heb je in ieder geval een voldoende. Maar ook, als de must-haves niet goed zijn heb je sowieso een onvoldoende.

3.1 Test-subroutines

Om te testen of je code goed werkt, vragen wij je om in ieder geval de volgende stukjes code om te zetten in je eigen programmeertaal.

3.1.1 Een dubbel-recursieve functie

```
bool even(unsigned int n);
bool odd(unsigned int n);

bool odd(unsigned int n){
    if(n==0){return false;}
    return even(n-1);
}

bool even(unsigned int n) {
    if(n==0){return true;}
    return odd(n-1);
}
```

[7] In de programmeertaal kan dit met bijvoorbeeld een expliciet return-statement, het resultaat van de laatste regel, of één vaste variabele/register die altijd aan het eind van de functie de return-waarde bevat.

[8] COBOL is momenteel een zeer gewenste taal omdat er nog veel legacy systemen zijn die erin geschreven zijn, maar de meeste COBOL-programmeurs zijn inmiddels met pensioen.

NB: wij weten ook wel dat dit simpeler kan, maar wij willen deze functies testen om te laten zien dat recursie werkt.

3.1.2 Een loopige functie

```
unsigned int sommig(unsigned int n){
    unsigned int result = 0;
    while(n>=1){
        result += n;
        n--;
    }
    return result;
}
```

Bij deze functie maken wij in C++ gebruik van een loopje. Je zult echter zien dat loopjes niet perse verplicht zijn in je programmeertaal. Daarom mag je deze functie omzetten naar een constructie met bijvoorbeeld `goto`-statements, of zelfs naar een recursieve variant voor lambda-calculus, als je taal geen expliciete loopjes ondersteunt. Indien je taal wél loopjes ondersteunt, willen we uiteraard wel graag dat je voor deze functie ook een loopje gebruikt.

3.2 Must-haves

Er zijn drie categoriën van must-haves. Criteria zijn eisen die wij stellen aan de stijl en manier van programmeren. Functionaliteiten zijn dingen die de interpreter moet kunnen. Inleveritems gaat over hoe en wat er precies moet worden ingeleverd.

3.2.1 Criteria

- De code is goed becommentariëerd.
- Er is een duidelijke README (zie ook Inleveritems).
- De code is geschreven in functionele programmeerstijl.
- De gekozen of ontworpen programmeertaal is Turingcompleet. Turing-compleetheid is aantoonbaar door aan te tonen dat de programmeertaal een Turing machine of Minsky (Register) machine kan implementeren, maar dit is vaak lastig. Je kan ook laten zien dat de programmeertaal in kwestie op zijn minst dezelfde functionaliteit kan bieden als een andere Turing-complete taal (zoals Brainfuck).
- Deze taal moet minstens of *loops* of *goto*-statments, of *lambda*-calculus ondersteunen.
- De taal mag géén Python, Brainfuck, C++ , of Assembler zijn.
- De code moet classes bevatten (voor bijvoorbeeld de Tokens en de AST), met inheritance.
- Object-printing functions voor elke class (via `__str__`)
- De code moet minstens één zelf-geschreven decorator (met -syntax) bevatten en gebruiken.
- Alle functions moeten type-annotated zijn (volgens het formaat in Hoofdstuk 4.4), zowel in commentaar (Haskell-type notatie) en in de functie zelf (Python-type notatie).
- Het gebruik van minstens 3 hogere-orde functies zoals: map, foldr, foldl, zipWith, zip, of andere (bijv. zelf-bedachte) hogere-orde functies. NB: de meeste standaard

hogere-orde functies zijn in Python geïmplementeerd en hoeven niet zelf geschreven te worden. Map zit bijvoorbeeld standaard zonder verdere imports in Python. *foldl* heet `reduce` in Python en zit in de `functools` library voor hogere-orde functies.

3.2.2 Functionaliteiten

- De interpreter moet minstens één functie per file kunnen supporten (mag ook meer)
- De interpreter moet de inputs van de functie kunnen meegeven (mag op de commandline, maar mag ook met een GUI).
- Een functie moet een andere functie kunnen aanroepen (als die in een andere file mag staan dan moet de interpreter die ook automatisch kunnen inladen)
- De interpreter moet het resultaat van de functie kunnen weergeven.

3.2.3 Inleveritems en -format

Ingeleverd moeten worden:

- De code van de interpreter
- Voorbeeldcode van je eigen programmeertaal, die in ieder geval de functies zoals beschreven in Paragraaf 3.1 implementeren.
- Een README met waar aan welke criteria en functionaliteiten voldaan zijn, hoe het programma gebruikt kan worden, welke libraries het programma nodig heeft (liefst zo min mogelijk), en waarom de taal Turing-compleet is. (Het format is te vinden op `https://docs.google.com/document/d/1X2lwEG-WXpHfp3GCPB6YgpgzUGDOO3Gjpec5NIg3imA/`).
- Een duidelijke argumentatie waarom de taal Turing-compleet is.
- Een video van 8 à 15 minuten waarin je a) de structuur taal die je ontworpen hebt of gebruikt kort toelicht b) de structuur van de code van de interpreter toelicht c) kort laat zien hoe de interpreter gebruikt wordt.

3.3 Should/Could-haves

Voor alles boven de 5.5 wil je graag extra functionaliteiten inbouwen. Hieronder staat een niet-uitputtende lijst van mogelijke extra functionaliteiten, met een maximum aantal punten. NB: je krijgt dus niet automatisch het maximum; meestal niet zelfs.

≤2pt Error-messaging - er is niets irritanter dan code die niet werkt met een stel vage foutmeldingen (ik zeg "segmentation fault") die je geen hint geven waar de fout eigenlijk zit. Help een programmeur uit de brand, en schrijf een goede error-handler waarmee een programmeur uit de voeten kan.

≤4pt Visualisatie kan erg interessant zijn, zeker als je bijvoorbeeld een editor hebt waarbij je de staat van het programma kan zien, of zelfs breakpunten kan zetten. Zie bijvoorbeeld de assembler-tool van Lennart Jenssen die vorig jaar is gemaakt: `https://github.com/Lennart99/ASM-interpreter`.

≤2pt Het is makkelijk om een Turing-complete taal te maken - zelfs Brainfuck is Turing complete (`https://softwareengineering.stackexchange.com/questions/315919/how-is-brainfuck-turing-complete`). Advanced language feautures boven bare-bone Turing-compleetheid (en function calls) leveren extra punten op.

≤1.5pt Het maken van een eigen programmeertaal is uitdagend, en levert sowieso 0.5 extra punt op. Wij kunnen nog 1 vol punt toekennen voor bijzonder creatieve dingen.

4 Opdracht 2: de compiler

In de tweede opdracht starten we met het werk dat in Opdracht 1 al gedaan is. Met name, de lexer en de parser kunnen volledig worden hergebruikt. Dit betekent natuurlijk niet dat je je lexer en parser precies zo moet laten als het was - voel je vrij om alle gewenste aanpassingen te maken.

Net als vorige opdracht, zijn er weer must-have and should-have criteria en functionaliteiten.

4.1 Must-haves

De must-have criteria zijn grotendeels identiek aan de criteria van de eerste opdracht. De functionaliteit is uiteraard wel anders.

4.1.1 Criteria

- De code is goed becommentariëerd.
- Er is een duidelijke README (zie ook Inleveritems).
- De code is geschreven in functionele programmeerstijl.
- De gekozen of ontworpen programmeertaal is Turingcompleet. Turing-compleetheid is aantoonbaar door aan te tonen dat de programmeertaal een Turing machine of Minsky (Register) machine kan implementeren, maar dit is vaak lastig. Je kan ook laten zien dat de programmeertaal in kwestie op zijn minst dezelfde functionaliteit kan bieden als een andere Turing-complete taal (zoals Brainfuck). Dit mag gekopiëerd zijn uit de eerste opdracht.
- Deze taal moet minstens of *loops* of *goto*-statments, of *lambda*-calculus ondersteunen. Indien je hieraan voldeed in de eerste opdracht voldoe je hier uiteraard ook aan bij de tweede opdracht.
- De taal mag géén Python, Brainfuck, C++ , of Assembler zijn.
- De code moet classes bevatten (voor bijvoorbeeld de Tokens en de AST), met inheritance. Indien je hieraan voldeed in de eerste opdracht voldoe je hier uiteraard ook direct aan bij de tweede opdracht.
- Object-printing functions voor elke class (via `__str__`)

!! Naast de code voor de compiler dient er een unit-test te zijn - geschreven in C++ - waarmee de compiler wordt getest. Dat wil zeggen, de output van de compiler moet meegecompileerd worden met deze unit-test en draaien op de Arduino. De unit-test toont aan dat de gecompilede code correct functioneert.

4.1.2 Functionaliteiten

- De compiler dient functies om te kunnen zetten naar één of meerdere Assembler voor de Cortex-m0.
- Er is een unit-test, geschreven in C++ .
- Er is een make-file die de (zelf-geschreven) compiler aanroept op de source-bestanden (in de gekozen programmeertaal) die deze source-bestanden compileert naar `.asm`-files. De make-file zorgt er vervolgens voor dat deze `.asm`-files door middel van een

standaard-compiler (zoals `gcc` of `g++`) worden meegecompileerd met de unit-tests tot een executable die vervolgens wordt uitgevoerd op een Arduino Due.

- De source-bestanden bevatten in ieder geval de equilavente code voor de voorbeelden uit Paragraaf 3.1, en nog minstens één eigen bedacht voorbeeld. Voor al deze source files zijn er unit tests.
- De unittests testen de code afdoende, en tonen daarmee aan dat de compiler goed werkt. De test moet dus uitputtend genoeg zijn, en eventuele randgevallen (en mogelijke slechte invoer) testen.

4.1.3 Inleveritems

- De code van de compiler
- De unit-test code in C++ , hierbij mag je gebruik maken van een unit-test library, maar dat hoeft niet (je mag het ook gewoon zelf in platte C++ -code schrijven).
- De source files in je eigen gekozen programmeertaal
- De makefile die de de source files d.m.v. je eigen compiler compileert naar `.asm`-files, en deze samen met de unittest compileert en draait op de Arduino Due.
- Een README met waar aan welke criteria en functionaliteiten voldaan zijn, hoe het programma gebruikt kan worden, welke libraries het programma nodig heeft (liefst zo min mogelijk).
- Een duidelijke argumentatie waarom de taal Turing-compleet is, if and only if dit was afgekeurd bij opdracht 1.
- Een video van 8 à 15 minuten waarin je a) de structuur van de code van de compiler toelicht b) uitlegt hoe de source voorbeelden vertaald worden naar `.asm`-files, en welke strategie je daarvoor gekozen hebt c) een demo, gebruik makend van je unittests.

4.2 Should/Could-haves

max 2pt Het showen van aanvullende functionaliteit door middel van unit-tests.

max 1pt Extra nieuwe functionaliteit, bovenop wat je bij opdracht 1 al had.

max 1.5pt Het genereren van commentaar in de `.asm`-files waardoor het voor de programmeur makkelijk terug te zoeken valt hoe zijn code wordt gecompileerd

max 4pt Optimalisatie van de compiler; uiteraard moet de code altijd werkend gecompileerd worden. Alle trucs die je echter toepast om de compiler efficiënter met registers om te laten gaan, of compactere code laten opleveren, leveren – mits goed gedocumenteerd in de README file – extra punten up. Het is hierbij handig om deze optimalisaties aan en uit te kunnen zetten, zodat het verschil duidelijk getoond kan worden (bijv. in de video).

max 1pt Uitbreiding van de error-handler van opdracht 1 naar de compiler.