Atte Vuorinen

Varjostinkielen toteuttaminen

Insinööri (AMK)

Tieto- ja viestintätekniikka

Syksy 2018

KAJAANIN
AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Tiivistelmä

Tekijä: Vuorinen Atte

Työn nimi: Varjostinkielen toteuttaminen

Tutkintonimike: Insinööri (AMK), tieto- ja viestintätekniikka

Asiasanat: Varjostinkieli, Leksikaalinen Analyysi, Kääntäjä Tekniikat

Työssä käsitellään yleisesti ohjelmointikielten rakennetta ja toteutuksessa tarvittavia tekniikoita. Ohjelmointikielet rakentuvat useista pienemmistä kokonaisuuksista, joita yleisesti ovat esiprosessointi, leksikaalinen analyysi, jäsentäjä ja koodin generointi.

Pääosin työssä keskitytään leksikaaliseen analyysiin ja jäsenten rakenteeseen ja toimintaan. Näiden tekniikoiden avulla voidaan esimerkiksi tulkita ohjelmointikieltä. Leksikaalisessa analyysissä sanoista muodostetaan saneita ja jäsentäjässä sanoista muodostetaan syntaksipuita käyttämällä apuna leksikaalisen analyysin saneita. Leksikaalinen analyysillä ja jäsentäjällä on kuitenkin samankaltainen rakenne ja toimintaperiaate.

Työn tavoitteena oli toteuttaa monipuolinen varjostinkieli. Kielen toteutus alkoi rakenteen ja avainsanojen suunnittelusta, ja jatkui leksikaalisen analyysin ja kääntäjän toteutukseen. Leksikaalisen analyysin avulla varjostinkielen rakennetta pystyi helposti laajentamaan ja jäsentämään. Lopuksi kääntäjä pystyi täydentämään varjostintiedoston tiedot käyttämällä leksikaalisen analyysin saneita.

Varjostinkielen toteutuksessa käytettiin C++-ohjelmointikieltä ja kolmannen osapuolen kirjastoja. Varjostintiedostojen täydentämisessä käytetty Mustache kaavainkirjasto mahdollisti JSON formaatin perusteella kaavaimien täydentämisen.

Työn lopputuloksena tuli toimiva monipuolinen varjostinkieli, jonka monipuolisuus saavutettiin käyttämällä kaavaintiedostoja. Näiden pohjalta lopulliset varjostintiedostot luodaan.

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Abstract

Author: Vuorinen Atte

Title of the Publication: Developing Shader Language

Degree Title: Bachelor of Engineering, Information and Communication Technology

Keywords: Shader Language, Lexical Analysis, Compiler Techniques

This Bachelor's thesis is about compiler techniques and the common structure of programming languages. Programming languages are often constructed by multiple processes which are preprocessing, lexical analysis, parser and code generation.

Mainly this thesis focuses on lexical analysis and parser which are mostly used for creating a programming language. This is because lexical analysis and parser are used for translating the source code. In a lexical analysis words are converted to tokens and in a parser, words are converted into parse trees with the lexical analysis tokens. Lexical analysis and parser often have similar structures.

The goal of this thesis was to develop a versatile shader language. The development was started by designing structure of the language along with keywords, and continued by implementing the lexical analysis and parser. Lexical analysis made it easier to implement new keywords and parse them. Lastly, compiler filled the shader information by using lexical analysis tokens.

The shader language was written in C++ programming language and by using 3rd party libraries. Mustache templating library made it possible to fill shader files by using JSON format.

As the result, a working versatile shader language was created. Versatility was obtained by using template files which are used for creating the final shader files.

Sisällys

1 Johdanto ... 1

2 Kääntäjätekniikka ... 2

2.1 Leksikaalinen analyysi ... 2

2.2 Kääntäjä ... 4

3 Varjostinkielet ... 6

3.1 GLSL- ja HLSL-varjostinkielet ... 7

3.2 Unity ShaderLab-varjostinkieli ... 8

3.3 Kirjastojen käyttö ... 9

4 Varjostinkielen toteutus ... 10

4.1 Varjostinkielen rakenne ... 11

4.2 Leksikaalinen analyysi ... 13

4.3 Kääntäjä ... 15

5 Tulokset ... 18

6 Yhteenveto ... 21

Lähteet ... 22

Symboliluettelo

Abstraktimuotoinen rakenne	Abstract Syntax Tree (AST), yksinkertaistettu syntaksipuu.
CBuffer	Constant buffer, varjostinkielissä käytetty muistialue globaaleille muuttujille.
CG	C for Graphics, Nvidian kehittämä varjostinkieli.
GLSL	OpenGL Shading Language, OpenGL rajapinnan kanssa käytettävä varjostinkieli.
HLSL	High-Level Shading Language, DirectX rajapinnan kanssa käytettävä varjostinkieli.
JSON	JavaScript Object Notation, JavaScript merkintätapa.
Jäsentäjä	Parser, ohjelma joka luo syntaksipuita tai abstraktimuotoisia rakenteita.
Kääntäjä	Compiler, ohjelmointikielen käännöksen suorittava ohjelma.
Leksikaalinen Analyysi	Lexical Analysic (Lexer), ohjelma joka käsittelee kielen rakennetta.
Konekieli	Machine code, prosessorin ymmärtämä kieli.
Sane	Token, merkkijonon rinnalle lisätty tieto.
Saneistaja	Tokenizer, leksikaalisen analyysissä saneitten luontia käsittelevä prosessi.
Syntaksi	Syntax, ohjelmointikielen rakenne ja säännöt.
Syntaksipuu	Parse tree, tarkkaan esitelty puurakenne kohteesta.
Tavukoodi	Bytecode, tietyn virtuaalikoneen tulkitsema kieli.
Uniform	Varjostinkielessä käsiteltävä globaali muuttuja.
Varjostin	Shader, näytönohjeimen suorittama ohjelma.
Välikoodi	Intermediate code, käännöksen välivaihe.

1 Johdanto

Projektin tavoitteena on luoda toimiva ja monipuolinen varjostinkieli, jonka avulla voidaan varjostinkieleen lisätä erilaisia ominaisuuksia taikka mukauttaa projektin tai grafiikkamoottorin tarpeita vastaavaksi. Varjostinkielen ominaisuuksien avulla voidaan kertoa grafiikkarajapinnalle tietoja, joiden avulla rajapinta pystyy asettamaan tarvittavat asetukset vastamaan varjostinta. Tällaisia ovat esimerkiksi läpinäkyvyys- ja syvyystiedot.

Projektin idea syntyi Unity-pelimoottorin ShaderLab-varjostinkielestä, jossa käytetään pohjana CG-varjostinkieltä ja sen lisäksi ShaderLab-kielen omia avainsanoja. Näillä avainsanoilla kerrotaan varjostimen vaatimia ominaisuuksia läpinäkyvyydelle, syvyydelle ja vastaaville tiedoille. Lopullinen varjostinkieli luodaan CG:n ja ShaderLab-varjostinkielen pohjalta tarvittavalle alustalle.

Nykyään ohjelmointikieliä on useita erilaisia eri tarkoituksiin ja niiden rakentajat koostuvat eri tavoin. Kuitenkin suurimassa osassa rakentajissa käytetään samoja tekniikoita. Näistä yleisempiä ovat leksikaalinen analyysi ja jäsentäjä, joiden avulla tulkitaan ohjelmointikieltä. Tästä syystä melkein kaikissa ohjelmointikielissä käytetään jonkin tasoista leksikaalista analyysiä.

2 Kääntäjätekniikka

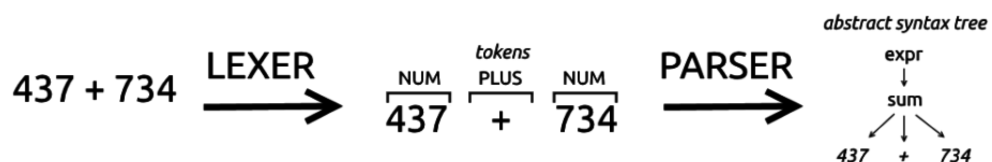
Ohjelmointikielten kääntäjät koostuvat useista pienemistä kokonaisuuksista, joista yleisimpiä ovat esiprosessointi, leksikaalinen analyysi, jäsennys ja koodin generointi. Kuitenkin jokaisessa ohjelmointikielessä nämä kokonaisuudet on eri tavoin toteutettu paremmin vastaamaan kielen käyttötarkoitusta.

2.1 Leksikaalinen analyysi

Leksikaalinen analyysi (Lexical Analysis) on ohjelmointikielten luonnissa käytetty erityinen jäsentäjä (Parser), joka tunnetaan myös nimellä sanestaja (Tokenizer). Leksikaalisen analyysin käyttötarkoituksena on tunnistaa ohjelmointikielen rakenne ja siihen liittyvät avainsanat, jotka se saneistaa saneiksi (kuva 1). Kyseisiä saneita hyödynnetään eri tavoin ohjelmointikielen jälkivaiheissa, kuten jäsentäjässä tai symbolitaulukossa. [1, s. 3]

Kuva 1. Leksikaalisen analyysin vaiheet

Jäsentäjissä saneita hyödynnetään jäsentämällä saneitten avulla syntaksipuita (Parse Tree) tai abstraktimuotoisia rakenteita (Abstract Syntax Tree / AST) tekstin sijaan tai tekstin täydennyksenä (kuva 2). Symbolitaulukossa saneiden avulla voidaan pitää paremmin kirjaa ohjelman tietotyypeistä, muuttujista, rakenteesta ja niiden rajauksista. [2.]



Kuva 2. Jäsentäjän rakenne [2]

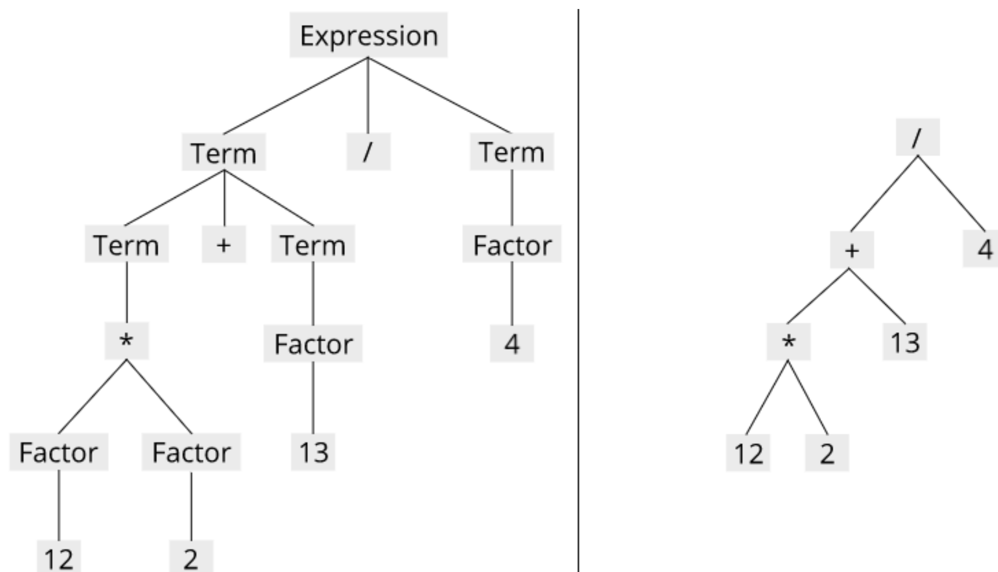
Leksikaalinen analyysi perustuu sääntöihin, joiden avulla saneet luodaan. C-ohjelmointikielessä leksikaalinen analyysi luo avainsanoista, tietotyypeistä, muuttujista ja komennoista symboliluettelon, jota käytetään myöhemmin jäsentäjässä ja kääntäjässä. [1, s. 2]

Saneista voidaan muodostaa erilaisia rakenteita, kuten puurakenteita tai S-lausekkeita eli symbolisia lausekkeita (kuva 3). Puurakenteiden avulla saneisiin voidaan lisätä helposti enemmän tietoa niiden sijainnista, jolloin saneita voidaan esimerkiksi rajata. S-lausekkeita käytetään enemmän tietorakenteena ja välivaiheena, jota voidaan tallentaa ja syöttää toisille ohjelmille.

```
(KEYWORD struct
    (IDENTIFIER 'A')
    (TYPE integer (IDENTIFIER 'value'))
)
```

Kuva 3. S-lauseke esimerkki

Syntaksipuussa tiedot pyritään esittämään mahdollisimman tarkkaan ja sisällöstä riippuen. Abstraktimuotoisessa rakenteessa sen sijaan pyritään esittämään tiedot mahdollisimman yksinkertaisesti ja universaalisesti. Syntaksipuuta jatkokäsittelemällä saadaan muodostettua abstraktimuotoisia rakenteita (kuva 4). [2.]



Kuva 4. Syntaksipuu ja abstraktimuotoisen rakenteen vertailu kaavasta (13 + 12 * 2) / 4

Leksikaalisessa analyysissä jäsentäjän tehtävänä on tunnistaa kirjaimia ja luoda niistä saneita. Tämän takia jäsentäjä voidaan helposti toteuttaa ylhäältä alas-menetelmällä (Top Down). Ylhäältä alas-menetelmässä jäsentäjä aloittaa jäsentämisen tekstin alusta ja jatkaa tekstin loppuun.

Isommissa jäsentäjissä sen sijaan käytetään alhaalta ylös-menetelmää (Bottom Up), joka on paljon tehokkaampi, mutta vaikeampi toteuttaa. Alhaalta ylös-menetelmässä jäsentäjä aloittaa jäsentämään tekstin lopusta ja jatkaa tekstin alkuun. Nykypäivänä alhaalta ylös-menetelmää käyttäviä jäsentäjiä luodaan erilaisten ohjelmien avulla, joiden jäsentämistä muokataan erillisten kielioppitietojen avulla (kuva 5). [2.]

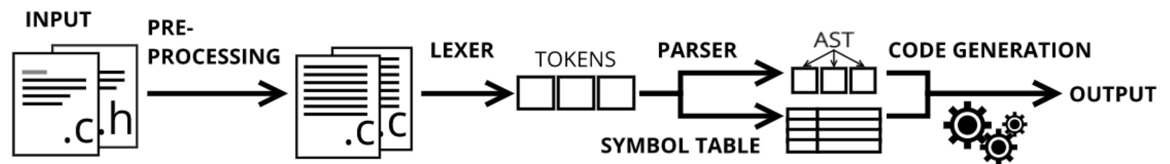
```
[Imports]
// ident is the name of the grammar

"COMPILER" ident
// this includes arbitrary fields and method in the target language (eg. Java)
[GlobalFieldsAndMethods]
// ScannerSpecification
CHARACTERS
  [..]
  zero          = '0'.
  zeroToThree   = zero + "123" .
  octalDigit    = zero + "1234567" .
  nonZeroDigit  = "123456789".
  digit         = '0' + nonZeroDigit .
  [..]

TOKENS
  ident         = letter { letter | digit }.
[..]
// ParserSpecification
PRODUCTIONS
// just a rule is shown
IdentList =
  ident <out int x>  (. int n = 1; .)
  {',' ident         (. n++; .)
  }                  (. Console.WriteLine("n = " + n); .)
  .
// end
"END" ident '.
```

Kuva 5. Coco/R kielioppi esimerkki [2]

2.2 Kääntäjä

Kääntäjä (Compiler) on ohjelmointikielen käännöksen hoitava ohjelma. Kääntäjä koostuu useista pienemmistä kokonaisuuksista, joita ovat esimerkiksi esiprosessointi, leksikaalinen analyysi, jäsentäjä ja koodin generointi. Koodin generointivaiheissa käytetään useasti välikoodia (Intermediate Code), jonka avulla lopullinen koodi luodaan. Välikoodin avulla tuotettavaa koodia voidaan paremmin optimoida ja jatkokäsitellä. [1, s. 2]

Kääntäjissä on erilaisia vaiheita, joita ovat esimerkiksi esiprosessointi, koodin generointi ja loppupää (kuva 6). Esiprosessointivaihetta ei kaikissa kääntäjissä ole, ja siinä esikäsitellään koodia esimerkiksi lisäämällä tai muokkaamalla sitä. Koodin generointivaiheeseen kuuluu leksikaalinen analyysi, jäsentäjä ja koodin generointi. Loppupäävaiheessa on optimointivaiheet ja kääntäjän lopputuloksen luonti. [1, s. 2]



Kuva 6. C-kääntäjän vaiheet [1, s. 2]

Kääntäjän lopputuotoksena on useasti suoritettava ohjelma tai kirjasto, mutta joissakin tapauksissa kääntäjä voi kääntää ohjelman toiselle ohjelmointikielelle. Ohjelmointikielten kääntäjät tuottavat ohjelman eri tavoin. Kuten C-ohjelmointikielessä, ohjelma käännetään ensin Assembly-ohjelmointikielelle ja sen kautta konekielelle (Machine Code).

Joissakin kielissä käytetään konekielen sijaan tavukoodia (Bytecode), jota suoritetaan toisen ohjelman kautta. Esimerkiksi, Java- ja Python-ohjelmointikielessä ohjelma käännetään ensin tavukoodiksi, joka se voidaan suorittaa virtuaalikoneessa [3]. Ohjelman suorittaminen virtuaalikoneessa mahdollistaa ohjelman kääntämisen sitä suorittaessa.

Välikoodin toteutuksessa yleensä käytetään triples- (two-address instructions), quads- (three-address instructions) ja postifx- (reverse-polish) notaatiota. Triples- ja quads-notaatiossa ensin tulee operaattori ja sen jälkeen muuttujat. Postfix-notaatiossa sen sijaan muuttujat tulevat ensin ja niiden jälkeen operaattori (kuva 7). [1, s. 447]

```
Triples     Quads       Postfix
(+ a b)   (+ a b c)     (a b +)
```

Kuva 7. Triples-, Quads- ja Postfix-notaatiot

3 Varjostinkielet

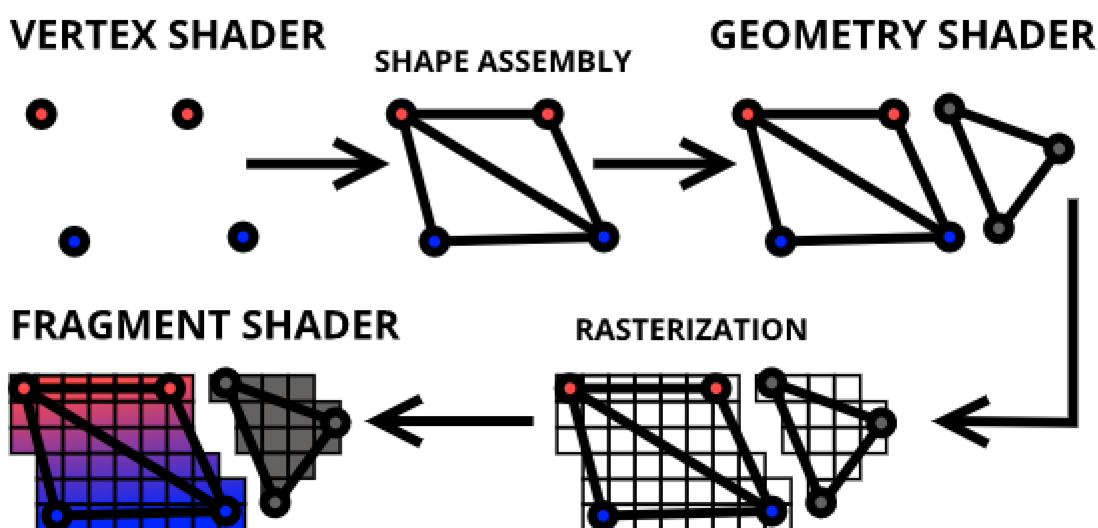
Varjostinkieli on tietyn grafiikkarajapinnan tukema ohjelmointikieli, jonka avulla vaikutetaan piirrettävien kappaleiden ominaisuuksiin. Kyseisiä ominaisuuksia voi olla kappaleiden kärkipisteiden paikka, rotaatio, koko, väri ym. Pääosin varjostinkielissä käsitellään vektori- ja matriisilaskentaa ja vältetään loogisia operaatioita.

Varjostinkieliä on useita, joista kaksi käytetyintä ovat GLSL- (OpenGL Shading Language) ja HLSL- (High-Level Shading Language) varjostinkielet. Kyseisten kielten suosio johtuu niiden tukemista grafiikkarajapinnoista.

OpenGL- ja Vulkan-grafiikkarajapinnat käyttävät GLSL varjostinkieltä. Sen sijaan OpenGL ES käyttää GLSL ES-versiota GLSL-kielestä. GLSL ES-versio on huomattavasti rajatumpi versio GLSL-kielestä ja tukee laajempaa laitekantaa. DirectX-grafiikkarajapinnan kanssa käytetään HLSL-varjostinkieltä.

Nvidian kehittämä CG-varjostinkieli tuki useampaa grafiikkarajapintaa kääntymällä suoraan GLSL- ja HLSL-varjostinkielille, mutta CG-varjostinkielen kehitys lopetettiin [4].

Varjostinkielissä on erityyppisiä varjostimia ja vaiheita, joista yleisempiä ovat kärkipistevarjostin, geometriavarjostin ja fragmenttivarjostin (kuva 8). Kärkipistevarjostimessa vaikutetaan piirrettävien kärkipisteiden ominaisuuksiin. Geometriavarjostimessa voidaan lisätä uusia kärkipisteitä, ja fragmenttivarjostimessa vaikutetaan pikseleihin.



Kuva 8. Varjostintyypit ja vaiheet

3.1 GLSL- ja HLSL-varjostinkielet

GLSL- ja HLSL-varjostinkielet toimivat eri tavoin, GLSL-kielessä on useita eri versioita ja käyttötapoja ja se vaikuttaa C-ohjelmointikieleltä. HLSL-kielessä sen sijaan on tarkemmin määritelty kielen käyttö ja se vaikuttaa enemmän C++-ohjelmointikielettä. Lisäksi molemmissa kielissä syöte- ja lähtöarvot on toteutettu eri tavoin, niin kuin globaalit muuttujat ja tietotyypitkin (taulukko 1). [5.]

	GLSL ES	GLSL	HLSL
Syötteet	attribute vec3 pos;	in vec3 pos;	struct In { float3 pos : POSITION; }
Lähdöt	varying vec3 color;	out vec3 color;	struct Out { float3 color : COLOR; }
Globaalit muuttujat	uniform	uniform	cbuffer
Tekstuurit muuttujat	sampler2D	sampler2D	Texture2D

Taulukko 1. GLSL ja HLSL, rakenteelliset erovaisuudet [5]

GLSL-kielessä globaalit muuttujat määritetään käyttäen uniform-avainsanaa. HLSL kielessä sen sijaan käytetään cbuffer- (constant buffer) avainsanaa globaalin tietorakenteen määrittämiseen. Tähän tietorakenteeseen määritellään muuttujat ja rekisteri (taulukko 2).

GLSL Variaatio 1	GLSL Variaatio 2	HLSL

uniform mat4 MVP;	struct Contants {
	    mat4 MVP;
	};
	uniform Contants contants;
		cbuffer constants : register(b0)
		{
		    float4x4 MVP;
		}

Taulukko 2. GLSL uniform ja HLSL cbuffer [5]

Muita eroavaisuuksia GLSL- ja HLSL- kielissä ovat muuttujatyypit, komentojen nimet ja matriisien arvojen järjestys. GLSL-kielessä matriisit täyttyvät oletuksena riveittäin ja HLSL-kielessä sen sijaan oletuksena sarakkeittain. [5.]

3.2 Unity ShaderLab-varjostinkieli

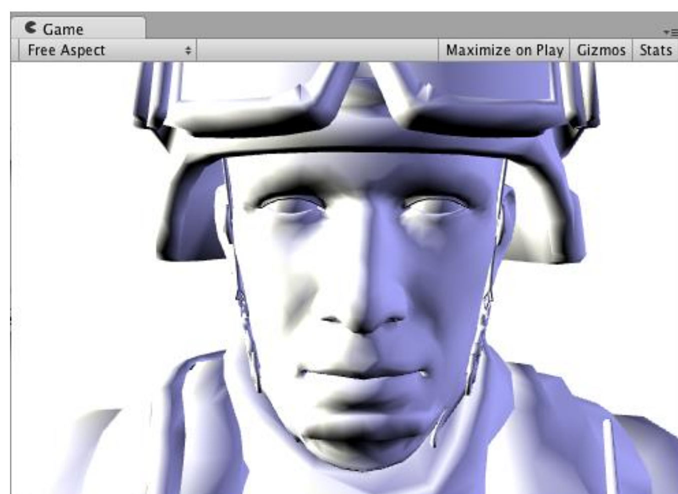
Unity-pelimoottorissa on käytössä Unityn kehittämä ShaderLab varjostinkieli. Kyseisessä varjostinkielessä varjostin kirjoitetaan CG- ja HLSL-koodilla ja kääntyy tarvittaessa muille varjostinkielille. ShaderLab-varjostin koostuu erilaisten avainsanojen avulla, jotka määräävät varjostimen ominaisuudet (kuva 9). [6.]

```
Shader "Example/Diffuse Simple"
{
  SubShader
  {
    Tags { "RenderType" = "Opaque" }
    CGPROGRAM
    #pragma surface surf Lambert

    struct Input
    {
      float4 color : COLOR;
    };

    void surf (Input IN, inout SurfaceOutput o)
    {
      o.Albedo = 1;
    }
    ENDCG
  }
  Fallback "Diffuse"
}
```

Program Info
Shader Program
Code

Kuva 9. ShaderLab, varjostinesimerkki [7]

ShaderLab-varjostinesimerkissä on käytössä-surface varjostin, joka on Unityn itse kehittämä varjostintyyppi. Kyseinen varjostin generoi kärkipiste- ja fragmenttivarjostimet. Lisäksi esimerkissä on käytössä Lambert-valaistusmalli ja varjostimen väri itsessään on valkoinen (kuva 10).



Kuva 10. ShaderLab-varjostinesimerkin tulos [7]

3.3 Kirjastojen käyttö

Varjostinkielen toteutuksessa on käytetty kolmannen osapuolen kirjastoja, joita ovat Mustache, Parson ja Zip. Mustache kirjasto on kaavainkirjasto, jonka avulla varjostimet täydennetään [8]. Parson-kirjaston avulla luetaan ja luodaan JSON-tiedostot ja tietorakenteet [9]. Zip-kirjaston avulla voidaan pakata ja purkaa ZIP-formaattiin kääntäjän luomat tiedostot.

Kaavaintiedostojen täydentämisessä käytetään Mustache-kaavainformaattia. Formaatissa tiedot täydennetään käyttämällä JSON-muuttujia, joita ovat yksittäiset, lista- ja objektimuuttujat. JSON-muuttujien käyttäminen tapahtuu aaltosulkeiden avulla (kuva 11).

`{{Program}} | {{ProgramType}}`
`{{#Blend}}`
`- {{.}}`
`{{/Blend}}`
	`vert | vertex`
	`- ScrColor`
	`- OneMinusScrColor`

Kuva 11. Mustache, muuttujaesimerkki

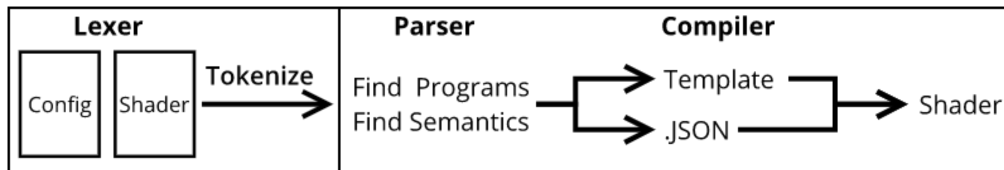
Mustache-formaatissa on mahdollista tehdä muuttujatarkistuksia. Tositarkistukset tehdään käyttämällä ristikkomerkkiä aaltosulkeissa ja epätodet tarkistukset tehdään käyttämällä sirkumfleksimerkkiä ja aaltosulkeissa (kuva 12). Tarkistuksissa tyhjät ja epätodet muuttujat palauttavat epätosiarvon ja muuten palauttavat tosiarvon.

`{{#List}} Not Empty List! {{/List}}` `{{^List}} Empty List! {{/List}}`	`Empty List!`

Kuva 12. Mustache-muuttujatarkistus esimerkki

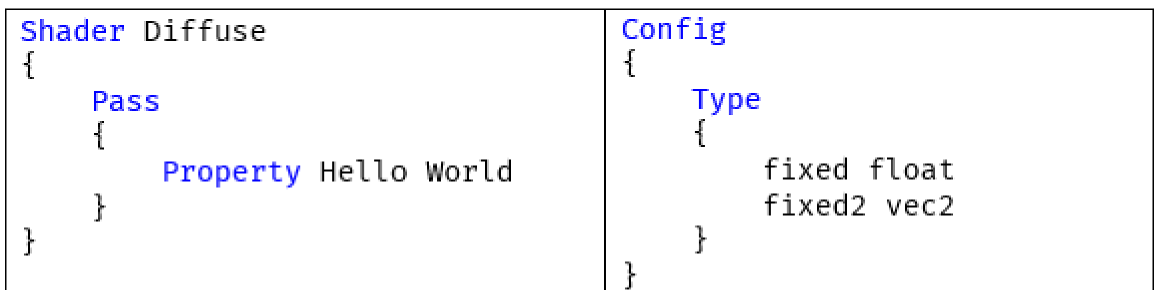
4 Varjostinkielen toteutus

Varjostinkielen toteutuksen tarkoituksena olisi olla mahdollisimman yksinkertainen, monipuolinen ja helppokäyttöinen. Varjostinkielen vaiheet ovat leksikaalinen analyysi, rakentaja ja JSON (JavaScript Object Notation) ja kaavaintiedoston (Template) täydentäminen (kuva 13). Kaavaintiedoston käyttötarkoituksena on mahdollistaa eri varjostinkielten kääntäminen ja niiden muokattavuus.



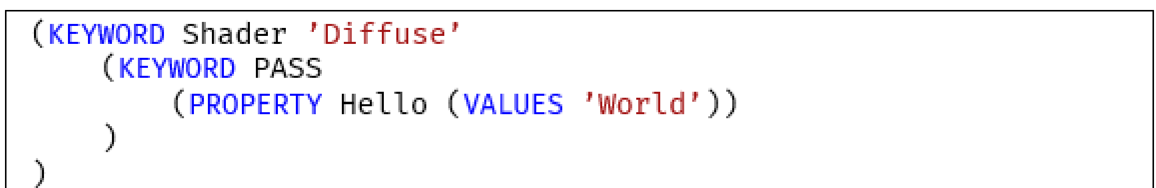
Kuva 13. Varjostinkielen vaiheet

Tässä varjostinkielessä on kaksi erilaista tiedostotyyppiä, asetustiedostot ja varjostintiedostot (kuva 14). Asetustiedostojen tarkoituksena on määrätä käännettävän varjostinkielen muuttujatyyppejä ja korvattavia avainsanoja. Varjostintiedossa sen sijaan tulisi määrittää itse varjostimet ja tarvittavat ominaisuudet, joita voidaan esimerkiksi hyödyntää grafiikkamoottorissa.



Kuva 14. Varjostinkieli ja asetusesimerkki

Kun varjostinkieli on syötetty leksikaalisen analyysin läpi, voidaan saneita kuvata esimerkiksi S-lausekkeen avulla (kuva 15). Tämän muotoilun avulla voidaan helposti suunnitella, miten varjostinkielen tietoa muotoillaan ja käytetään.



Kuva 15. S-lauseke-esimerkki

4.1 Varjostinkielen rakenne

Projektin varjostinkieli toteutuu kahdesta erillisestä asetuksesta. Ensimmäinen asetus on tarkoitettu asetuksien lukemiseen ja toinen on varjostimen lukemiseen. Asetukset määräävät varjostimen tietotyypit, tekstuurit ja korvattavat avainsanat ja tulee aina aloittaa ”Config”- tai ”Target”-avainsanalla (taulukko 3).

Avainsana	Muuttujat	Kuvaus
Config	[Nimi]	Asetukset
Target	<Nimi>	Asetuskohde
Match	<Nimi> <Arvo>	Korvattava termi
Type	<Nimi> [Arvo]	Tietotyyppi
Texture	<Nimi> [Arvo]	Tekstuurityyppi

Taulukko 3. Asetustiedoston avainsanat

Varjostimessa määritellään varjostimien ominaisuudet ja toiminnallisuus. Varjostintiedosto alkaa ”Shader”-avainsanalla ja rakentuu tasojen avulla, jotka määräytyvät ”Pass”-avainsanan avulla (taulukko 4). Varjostintasoissa on vaatimuksena määritellä tasoille rajaukset käyttäen aaltosulkeita.

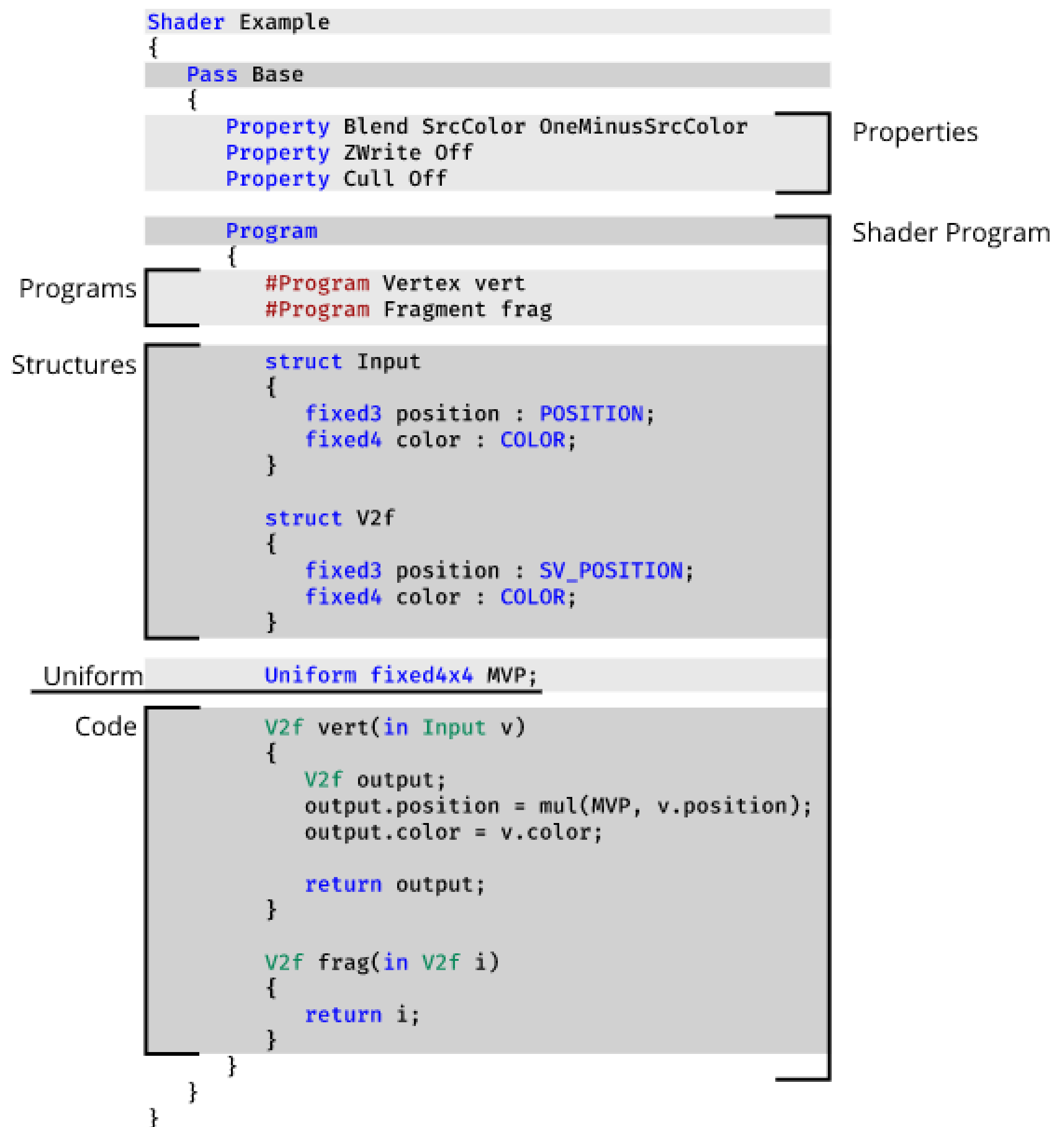
Avainsana	Muuttujat	Kuvaus
Shader	<Nimi>	Varjostimen nimi
Pass	[Nimi]	Varjostintaso
Property	<Nimi> [Arvot…]	Muuttuja
Property	<Nimi> **{** <Muuttujat> **}**	Muuttujajoukko
Program		Varjostintason ohjelma
#Include	<Tiedosto>	Tiedoston sisällytäminen
#Program	<Tyyppi> <Nimi>	Varjostinohjelman määritelmä
Struct	<Nimi>	Tietorakenne

Taulukko 4. Varjostintiedoston avainsanat

Varjostinkielen avainsanoilla on erilaisia vaatimuksia, joita ovat esimerkiksi nimi, muuttuja- arvot tai rajaus. Muuttujatyyppeinä ovat yksittäiset muuttujat, listamuuttujat ja joukkomuuttujat. Joukkomuuttujat koostuvat rajauksen sisällä olevista muuttujista.

”Property”-avainsanan muuttujilla on erilaisia toimintotapoja, jotka muuttavat niiden JSON- esitystapoja. Oletuksena muuttujat määritellään yksitellen. Kuitenkin muuttujia voi määritellä listana, jolloin JSON-muodossa ne esitetään listana. Tarvittaessa muuttujia voi esittää myöskin rajauksien avulla objektina, jolloin JSON-muodossa ne esitetään JSON-objektina.

Varjostinohjelman koodiosio tulee ”Program”-avainsanan määrittelemän rajauksen sisälle. Varjostinohjelmat pitää määritellä erikseen käyttämällä ”#Program”-avainsanaa, johon laitetaan ohjelman tyyppi ja komennon nimi. Ohjelman määrityksen avulla kääntäjä osaa etsiä tarvittavan varjostinkomennon ja sille sisääntulo- ja ulostuloarvot (kuva 16).



Kuva 16. Varjostinesimerkki

Jotta GLSL- ja HSLS-kielten globaalit muuttujat olisivat tuettuja, ne pitää erikseen määritellä ”CBuffer”- tai ”Uniform”-avainsanalla. Riippuen kohdekielestä ”CBuffer”-arvojen rekisteri voidaan tarvittaessa ohittaa ja määritellä ne pelkästään ”Uniform”-avainsanalla.

Varjostimen aikaisemmilla tasoilla voidaan määritellä oletusarvoja. Esimerkiksi ”Property”-avainsanan tiedot periytyvät alemmille varjostintasoille. Kuitenkin oletusarvoja voi uudelleen määritellä alemmilla tasoilla, jolloin niiden arvot korvaavat aikaisemmat arvot.

Varjostinkielen rakenteessa on otettu jonkin verran vaikutteita Unityn ShaderLab-kielestä. Isoimpia samankaltaisuuksia ovat varjostimen tasojen rakenne ja varjostinohjelman määritteleminen. ShaderLab-kielen ohjelma ja sen tietojen määrittäminen tapahtuu ”#pragma”-avainsanan avulla. Toteutetussa varjostinkielessä varjostinohjelma määritellään ”#Program”-avainsanan avulla. Kyseisen määritelmän avulla kääntäjä osaa etsiä varjostinohjelman komennon.

4.2 Leksikaalinen analyysi

Tässä projektissa leksikaalinen analyysi perustuu välilyönteihin ja rivivaihtoihin niin kuin Python-ohjelmointikielessä. Tämä pitää kielen rakenteen ja toteutuksen yksinkertaisena. Lisäksi valmiiksi suunniteltu rakenne auttaa selventämään leksikaalisen analyysin tarpeita. Tässä tapauksessa suurimmat poikkeukset tapahtuvat varjostinohjelman rajauksessa, jossa suurin osa sisällöstä jätetään saneistamatta.

Ensimmäiseksi leksikaalinen analyysi pyrkii erottelemaan rivissä olevat sanat välilyöntien perusteella, jonka jälkeen sanoista saadaan tietyn tyyppisiä saneita ja saneet lisätään saneen puurakenteeseen (kuva 17).

```
Token token = CreateToken()
while (not EndOfLine(buffer))
      String word = GetWord(buffer)
      UpdateToken(token, word)
AddToken(token)
```

```
if word == '{'
   scope = token
else if word == '}'
   scope = token.scope
```

Kuva 17. Pseudokoodisaneiden luonti ja rajaus

Saneitten rajaukset määräytyvät aaltosulkeiden ja viimeisimmän saneen perusteella. Tämän avulla saadaan saneista muodostettua puurakenne, jota voidaan hyödyntää rajauksissa ja kääntäjässä. Rajaukset on toteutettu siten, että saneilla on tieto niiden nykyisestä rajauksesta, jonka avulla voidaan palata rajauksessa taaksepäin ja rajausta voidaan sisentää vaihtamalla rajauksessa käytettyä sanetta (kuva 17). Kaikki avainsanat eivät kuitenkaan tue rajauksia ja osalle ne ovat välttämättömiä. Rajauksien avulla voidaan määrittää useampia varjostimia yhteen varjostintiedostoon.

Saneiden tunnistamisessa käytetään useita listoja, joihin on määritelty tiettyjä avainsanoja niiden ominaisuuksien perusteella. Tämän avulla saneet saavat tunnisteen, joita ovat esimerkiksi avainsana, tietotyyppi ja muuttaja. Jos saneella on jo tunniste, saneeseen lisätään ensin nimi ja sen jälkeen tarvittaessa muuttuja-arvoja (kuva 18).

```
if token.type == None
      token.type = FindType(word)
else if IsEmpty(token.name)
      token.name = word
else
      AddValue(token, word)
```

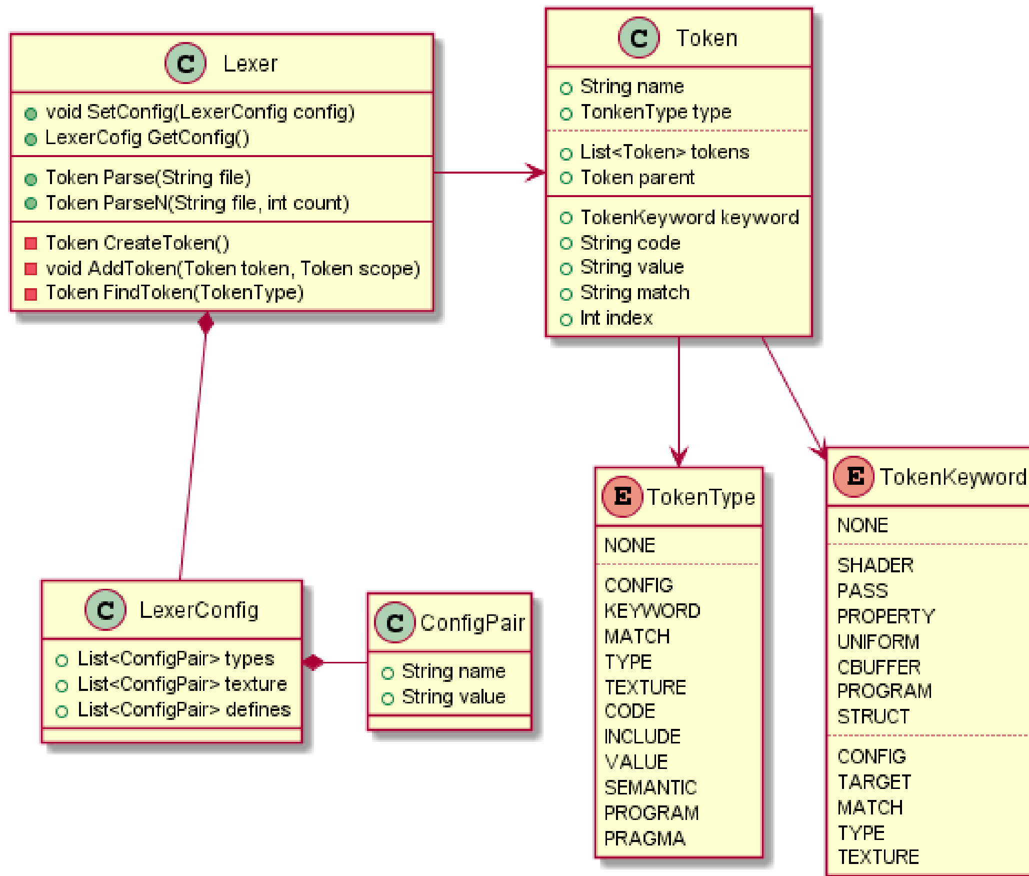
```
function FindType(word)
  if IsKeyword(word)
     return Keyword(word)
  else if IsType(word)
     return Type(word)
  return None
```

Kuva 18. Pseudokoodisaneiden tunnisteet ja tiedot

Saneiden tunnisteiden avulla leksikaaliseen analyysiin voidaan lisätä erilaisia sääntöjä, kuten onko saneella oltava nimi ja saako saneella olla rajausta. Nämä säännöt voidaan helposti määrittää eräänlaisena matriisina, jossa saneen tunnisteen avulla etsitään saneeseen liittyvät säännöt. Tämän avulla sääntöjä on todella helppo lisätä ja muokata.

Leksikaalisen analyysin saneet sisältävät nimen, tunnisteen ja tunnisteeseen liittyvät muuttujat. Lisäksi saneisiin on lisätty puurakenteeseen tarvittavat tiedot, joita ovat lista rajauksen sisälle kuuluvista saneista ja saneen omistaja (kuva 19). Asetukset ovat sen sijaan lista muuttujapareista. Näitä muuttuja pareja käytetään varjostimen saneistuksessa ja myöhemmin kääntäjässä muuttujatyyppien korvaamisessa (kuva 19).

Saneen tunnistemuuttujia ovat avainsanat, ohjelmaosiot, muuttuja arvot ja korvattavat sanat. Nämä muuttujat on määritelty kaikille saneille, ja niistä käytetään vain yhtä riippuen tunnisteesta. Esimerkiksi avainsanasaneet sisältävät tiedon kyseisestä avainsanasta ja muuttujasaneet sisältävät muuttujan nimen ja arvot (kuva 19).



Kuva 19. Leksikaalisen analyysin UML-kaavio

Jos saneelle ei löydetä sopivaa tunnistetta, niin saneistaja ymmärtää sen muuksi varjostinkoodiksi. Nämä varjostinkoodit lisätään saneeseen tekstinä ja omana tunnisteena, näitä saneita käytetään myöhemmin kääntäjässä.

4.3 Kääntäjä

Kääntäjä perustuu erilaisiin vaiheeseen, jotka ovat asetuksien lukeminen, varjostimen leksikaalinen analyysi, saneiden jäsentäminen, varjostintasojen kaavoittaminen ja varjostintiedoston jälkikäsittely ja pakkaaminen.

Ensimmäisessä vaiheessa kääntäjä saneistaa oletusasetukset ja sen jälkeen käännettävien varjostinkielten asetukset. Tämän jälkeen kääntäjä alkaa saneistamaan varjostintiedostoja, käyttäen saneistettuja asetuksia.

Seuraavaksi kääntäjä jäsentää varjostimesta yhteiset ohjelmat ja ominaisuudet. Tämän jälkeen kääntäjä jäsentää varjostimen tasot. Jokaisesta tasosta kääntäjä jäsentää varjostinohjelmat ja niiden syöte- ja tuloarvot. Näiden tietojen perusteella kääntäjä luo väliaikaisen JSON tiedoston (kuva 20), joka syötetään käännettävälle varjostintiedoston kaavaintiedostolle. Kaavaintiedoston täydentäminen suoritetaan jokaiselle ohjelmalle, tasolle ja kääntäjän kohteille.

```
{
    "Blend": ["SrcColor", "OneMinusSrcColor"],   Properties
    "ZWrite": "Off",
    "Cull": "Off",

    "InputType": "Input",                        Input Info
    "Input": [
        {
            "Type": "fixed3",
            "Name": "position",
            "Semantic": "POSITION"
        },
        {
            "Type": "fixed4",
            "Name": "color",
            "Semantic": "COLOR"
        }
    ],

    "OutputType": "V2f",                         Output Info
    "Output": [
        {
            "Type": "fixed3",
            "Name": "position",
            "Semantic": "SV_POSITION"
        },
        {
            "Type": "fixed4",
            "Name": "color",
            "Semantic": "COLOR"
        }
    ],

    "Program": "vert",                           Program Info
    "ProgramType": "vertex",
    "Uniforms" : [{...}],
    "Code": "..."
}
```

Kuva 20. Kaavaintiedoston täydentävä JSON-sisältö

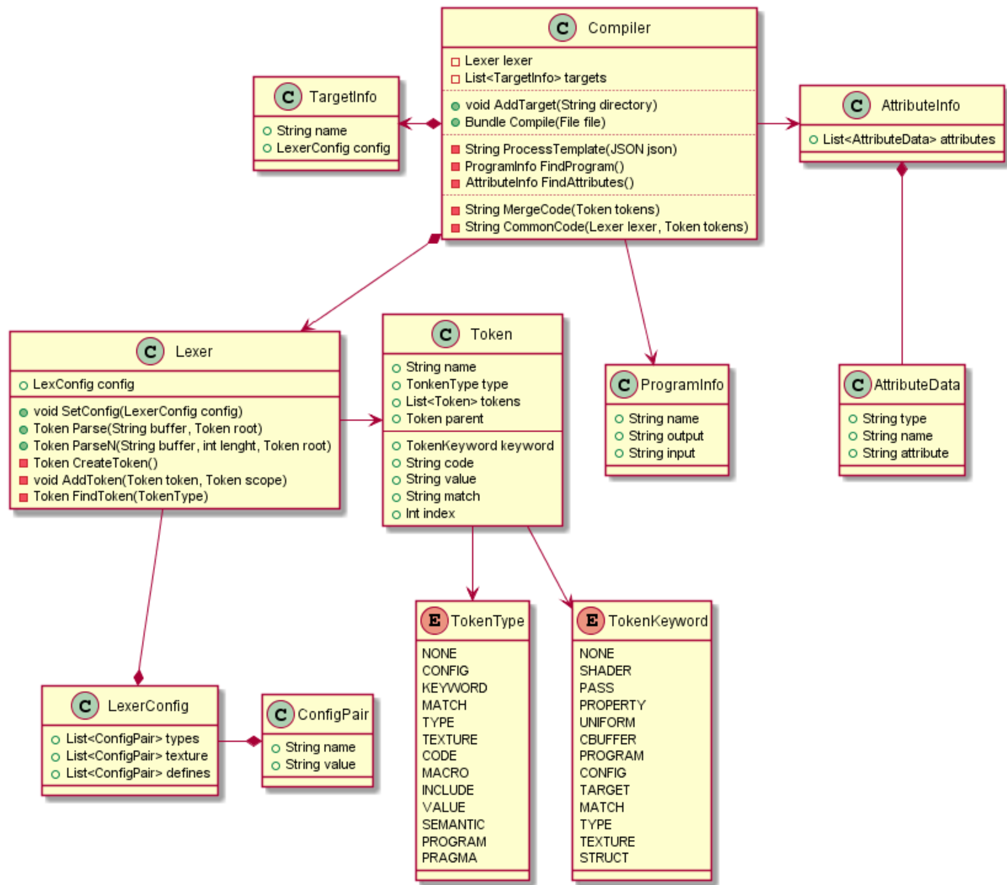
Jäsentäminen tapahtuu kääntäjässä saneitten avulla, jossa käännetyistä saneista etsitään rekursiivisesti tiettyjä tietoja (kuva 21). Näitä tietoja voi olla esimerkiksi saneen tyyppi tai avainsana. Lisäksi saneiden jäsentäminen voidaan rajata varjostimen tason sisä- tai ulkopuolelle.

```
function FindPasses(root)
  if root.keyword == PASS
     Append(root)
  for each (token in root.tokens)
     FindPasses(token)
```

Kuva 21. Pseudokoodi, rekursiivinen jäsennys

Viimeisessä vaiheessa kääntäjä luo varjostintiedoston pohjalta JSON-tiedoston, jossa on kaikki varjostimeen liittyvä tieto. Näitä tietoja ovat esimerkiksi varjostimen nimi, ohjelmat, ominaisuudet ja globaalit muuttujat. Kyseisiä tietoja voidaan myöhemmin hyödyntää varjostimen käytössä. Lopuksi kääntäjä pakkaa käännetyt varjostimet ja JSON-tiedoston ZIP-formaattiin.

Kääntäjä hyödyntää leksikaalista analyysia ja pitää sen tietoja tallessa. Käännettävien kohteiden tiedot kääntäjä pitää listassa, jossa on kohteen nimi ja asetukset (kuva 22). Myöhemmin varjostimen kääntövaiheessa kääntäjä jäsentää varjostinohjelman tiedot, joita ovat ohjelman nimi, syöte- ja tuloarvot. Varjostinohjelman syöte- ja tuloarvojen perustella jäsennetään muuttuja-arvojen tiedot (kuva 22).



Kuva 21. Kääntäjän UML-kaavio

5 Tulokset

Aikaisemman varjostimen pohjalta (kuva 16) saadaan lopullinen JSON-tiedosto (kuva 23), jossa ovat varjostimeen ja varjostinohjelmien liittyvät tiedot. Näistä tiedoista löytyy aina varjostimen nimi, -ohjelmat, -tasot ja ominaisuudet. Lisäksi jokaisella ohjelmalla on erikseen ohjelman tarvittavat tuloarvot.

```
{
    "Shader": "Example",
    "Programs": ["vertex","fragment"],
    "PassCount": 1

    "Passes": [
        {
            "Name": "Base",
            "Index": 0,
            "Uniforms": [{...}],
            "Blend": ["SrcColor","OneMinusSrcColor"],
            "ZWrite": "Off",
            "Cull": "Off",

            "vertex": {
                "GLSL": "GLSL/Example_Base.vertex",
                "Input": [
                    {
                        "Name": "position",
                        "Type": "fixed4",
                        "Semantic": "POSITION"
                    },
                    {
                        "Name": "color",
                        "Type": "fixed4",
                        "Semantic": "COLOR"
                    }
                ]
            },

            "fragment": {
                "GLSL": "GLSL/Example_Base.fragment",
                "Input": [
                    {
                        "Name": "position",
                        "Type": "fixed4",
                        "Semantic": "SV_POSITION"
                    },
                    {
                        "Name": "color",
                        "Type": "fixed4",
                        "Semantic": "COLOR"
                    }
                ]
            }
        }
    ],
}
```

Shader Info

Pass Info

Properties

Vertex program

Fragment program

Kuva 23. Varjostin, JSON-tiedosto

Lisäksi aikaisemmasta varjostimesta (kuva 16) saadaan GLSL-varjostin kaavaintiedoston avulla (kuva 24). Kyseisellä GLSL kaavaintiedostolla saadaan generoitua ”vertex”- ja ”fragment”-varjostimet.

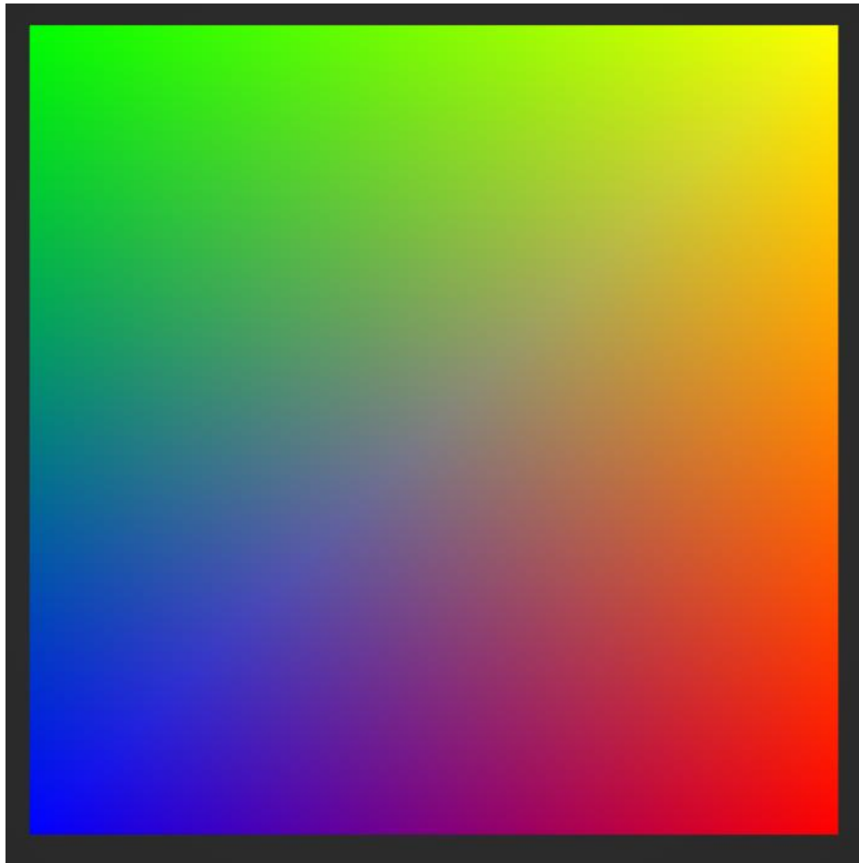
GLSL-kaavaintiedostossa täydennetään ensin tarvittavat muuttujatiedot ja ominaisuudet varjostimelle. Tämän jälkeen varjostimeen lisätään yleinen koodi. Yleisen koodin jälkeen määritellään varjostinohjelman syöte- ja lähtömuuttujatyypit ja varjostinohjelman koodi. Lopuksi määritellään GLSL-varjostinkielen omat vaatimukset, tässä tapauksessa pääohjelma ”main”- ja GLSL-kielen mukaiset syötteet ja lähdöt (kuva 24).

```
{{#fragment}} precision mediump float; {{/fragment}}

{{Defines}}

// Sets program define
#define {{ProgramType}} 1

{{#vertex}} #define INPUT_KEYWORD attribute {{/vertex}}
{{^vertex}} #define INPUT_KEYWORD varying {{/vertex}}

// Common code
{{CommonCode}}

// Program Input type
struct {{InputType}} {
   {{#Input}} {{Type}} {{Name}}; {{/Input}}
};

{{#OutputStruct}}
   // Program Output type
   struct {{OutputType}} {
      {{#Output}} {{Type}} {{Name}}; {{/Output}}
   };
{{/OutputStruct}}

// Program code
{{Code}}

// GLSL style input
{{#Input}}
   INPUT_KEYWORD {{Type}} {{InputType}}_{{Name}};
{{/Input}}

{{#OutputStruct}}
   // GLSL style output
   {{#Output}}
   varying {{Type}} {{OutputType}}_{{Name}};
   {{/Output}}
{{/OutputStruct}}

void main()
{
   // Program input
   {{InputType}} i;

   // Populate input i
   {{#Input}}
      i.{{Name}} = {{InputType}}_{{Name}};
   {{/Input}}

   // Output
   {{OutputType}} o = {{Program}}(i);

   {{#OutputStruct}}
   // Set GLSL output values
      {{#Output}}
         {{OutputType}}_{{Name}} = o.{{Name}};
      {{/Output}}
   {{/OutputStruct}}

   {{#vertex}} gl_Position = o.position; {{/vertex}}
   {{#fragment}} gl_FragColor = o; {{/fragment}}
}
```

```
precision mediump float;

#define fixed4x4 mat4
#define fixed4 vec4
#define fixed3 vec3
#define fixed2 vec2
#define fixed float

// Sets program define
#define fragment 1
#define INPUT_KEYWORD varying

// Program Input type
struct V2f {
   fixed4 position;
   fixed4 color;
};

// Program code
#ifdef vertex
...
#endif

#ifdef fragment
fixed4 frag(in V2f i)
{
   return i.color;
}
#endif

// GLSL style input
INPUT_KEYWORD fixed4 V2f_position;
INPUT_KEYWORD fixed4 V2f_color;

void main()
{
   // Program input
   V2f i;

   // Populate input i
   i.position = V2f_position;
   i.color = V2f_color;

   // Output
   fixed4 o = frag(i);

   // Fragment color output
   gl_FragColor = o;
}
```

Kuva 24. GLSL Mustache-kaavaintiedosto ja GLSL-varjostinesimerkki

GLSL-varjostinesimerkissä (kuva 24) fragmenttivarjostimen tuottama väri perustuu kärkipisteiden väreihin (kuva 25). Kyseinen värin siirtäminen tapahtuu ”frag”-komennossa, jossa syötteen väri palautetaan. Lopullinen lähtöväri asetetaan ”gl_FragColor”-muuttujaan.



Kuva 25. GLSL-varjostinesimerkin tulos

Lopputuloksena toteutettu varjostinkieli on tarpeeksi monipuolinen kääntämään eri varjostinkielille käyttämällä kaavaintiedostoja. Kuitenkin varjostinkielelle löytyy vielä parannettavaa, kuten tarkemmat virheilmoitukset ja esi- ja jälkiprosessointi varjostintiedostoille ja kaavaintiedostoille. Varjostinkielen toteutus onnistui ilman suurempia ongelmia ensimmäiseksi kieleksi.

6 Yhteenveto

Varjostinkielen toteutusta varten kannattaa suunnitella tarkkaan kielen tarkoitus ja sen rakenne, jotta kielen rakenteesta ei tulisi liian monimutkaista ja laajaa. Toteutetussa varjostinkielessä kieli rakentuu kahdesta erillisestä tilasta, joista varjostimen tietotyypit ja muut tiedot määritellään asetustiedostojen avulla ja varjostin omalla varjostintiedostolla.

Varjostintiedosto koostuu tasojen avulla, joita määritellään käyttämällä ”shader”- ja ”pass”-avainsanoja. Kielen avainsanoilla on määritelty tiettyjä sääntöjä, joiden avulla kielen toteutus yksinkertaistuu. Näitä sääntöjä ovat aaltosulkeiden käyttö ja avainsanojen vaaditut tiedot.

Varjostinkielen rakenteen ollessa selkeä on leksikaalisen analyysin toteuttaminen paljon yksinkertaisempaa ja mahdollistaa helpon koodin uudelleen käyttämisen. Tässä toteutuksessa leksikaalinen analyysi määrittelee riveistä yksittäiset saneet ja lisää ne puurakenteeseen myöhempään käyttöön.

Varjostimen kääntäminen alkaa asetustiedostojen leksikaalisella analyysillä ja jatkuu varjostimen leksikaalisella analyysillä käyttäen asetustiedostoja. Varjostimen saneista jäsennetään varjostinohjelmien tiedot, ja lopulta varjostin luodaan käyttämällä kaavaintiedostoa. Viimeiseksi kääntäjä pakkaa käännetyt varjostimet ja varjostimen tiedot ZIP-tiedostoon.

Lähteet

1 Compiler Design In C kirja. Haettu osoitteesta: https://holub.com/compiler/

2 Federico Tomassetti Parsing In C#: Tools and Libraries. Haettu osoitteesta: https://tomassetti.me/parsing-in-csharp/

3 Python Glossary Bytecode. Haettu osoitteesta: https://docs.python.org/3/glossary.html#term-bytecode

4 Nvidia CG. Haettu osoitteesta: https://developer.nvidia.com/cg-faq

5 GLSL-to-HLSL Reference. Haettu osoitteesta: https://docs.microsoft.com/en-us/windows/uwp/gaming/glsl-to-hlsl-reference

6 Unity ShaderLab Manual. Haettu osoitteesta: https://docs.unity3d.com/Manual/SL-Shader.html

7 Unity ShaderLab Surface Examples. Haettu osoitteesta: https://docs.unity3d.com/Manual/SL-SurfaceShaderExamples.html

8 Github Kainjow Mustache Library. Haettu osoitteesta: https://github.com/kainjow/Mustache

9 Github Kgabis Parson Library. Haettu osoitteesta: https://github.com/kgabis/parson

10 Github Kuba Zip Library. Haettu osoitteesta: https://github.com/kuba--/zip