eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Informatika Ingeniaritzako Gradua

Konputazioa

Gradu Amaierako Lana

Fleet programazio lengoaia

Egilea

Aritz Lopez Lajas

Zuzendaria

Nerea Ezeiza Ramos

2018

Laburpena

Fleet sintaktikoki C lengoaiaren antzekoa den helburu orokorreko programazio lengoaia inperatiboa da. Honen konpiladoreak LLVM eta ANTLR4 liburutegiak erabilitzen ditu kodea hainbat plataforma desberdinetarako konpilatu ahal izateko.

Gaien aurkibidea

Irudien aurkibidea

1. KAPITULUA

Sarrera

Atal honetan Fleet programazio lengoaia sortzeko motibazioa deskribatzen da. Jarraian, inspirazio moduan erabilitako antzeko beste proiektu batzuk aipatuko dira. Azkenik, proiektu honen helburua zein den finkatuko da, eta irismena zehaztuko da.

1.1 Motibazioa

Programatzen duen edozeinek bezala, erabili ditudan lengoaia guztietan gustuko ez ditudan ezaugarriak aurkitu ditut. Horregatik, programatzen hasi nintzenetik, nik diseinatutako programazio lengoaia bat sortu nahi izan dut. Nahi hori aurretik bete ez izateko arrazoi nagusia konpiladoreen funtzionamenduari buruzko ezagutza eza izan da, baina fakultatean *Konpilazioa* ikasgaia ikasi ondoren, honelako proiektu bat garatzeko oinarrizko ezagutzak lortu nituela kontsideratu nuen.

Nahiz eta proiektu hau garatu ahal izateko *Konpilazioa* ikasgaiaren edukiak ezinbestekoak diren, bertan erabilitako tresnak ez erabiltzea erabaki dut, hainbat arrazoirengatik. Alde batetik, tresna horiek 1980ko hamarkadan sortu ziren, eta garai horretako ordenagailu eta lengoaien ezaugarrietara egokituta daude. Egia da tresna hauek erabiltzen jarraitzen direla, baina ez dituzte tresna berriek dituzten ezaugarriak eskaintzen (adibidez, gramatika idazteko malgutasuna eta integratutako garapen inguruneetan gramatika koloreztatu eta iradokizunak erakusteko aukera). Tresna hauetan, orokorrean, analisi lexiko eta sintaktikoa banatzen dira, batzuetan programa desberdinak erabiltzera behartuz. Tresna modernoek lexikoia eta gramatika fitxategi berean definitzeko aukera emateaz gain, bi prozesuak bateratu egiten dituzte, garatzailearen lana sinplifikatuz.

1.2 Inspirazioa

Proiektua garatzen hasi baino lehen, helburu antzekoak izan ditzaketen eta inspirazio moduan erabil ditzakedan beste proiektu batzuk bilatu ditut. Ondorengo bi programazio lengoaiak C edo C++-etik eratorriak dira, baina bere sortzaileek gustuko ez zituzten ezaugarriak aldatuta.

1.2.1 D

ALGOL, *B / BCPL* eta *C* programazio lengoaien ondoren, ordena alfabetikoan datorren hurrengoa *D* da. Walter Bright-ek sortutako programazio lengoaia honek, bere aurrekaria den C++-en ezaugarri anitz mantentzen ditu, baina programazioa erraztu dezaketen beste hainbeste ere baditu. Hala nola, *garbage collection* edo memoria-kudeaketa automatikoa, taula elkarkorrak (*Associative array*), eta taulen zatiketa (*array splicing*).

Gainera, D programazio lengoaiak hiru *front-end* desberdin ditu: DMD, GDC eta LDC. Lehenengoa erreferentziazko konpiladorea da, eta bi plataformatarako *back-end* propio bat ere badu integratuta. GDC GCCren *front-end* bat da, eta GCCk konpila dezakeen plataforma gehienetarako konpila dezake. Azkena, LDC, LLVM erabiltzen du *back-end* moduan, eta C++-en idatzita dago.

1.2.2 Zig

Andrew Kelley garatzen ari den lengoaia bat da. Zig-ek Cren ezaugarri antzekoak ditu, baina arazo lodi batzuk konpontzen ditu. Adibidez, erroreak tratatzeko datu-mota berezi bat du, *null* erakusleak saihesteko mota hutsak erabili ditzake. Gainera, C-rekin guztiz bateragarria da, bi norabideetan.

Bi konpiladore ditu: lehenengoa, Cz idatzita dagoena. Ondoren, hau erabilita, Zig-ez idatzitako konpiladorea konpilatu daiteke. Azkenik, honela lortutako konpiladorea erabilita, berriro ere Zig-ez idatzitako konpiladorea konpilatzen da, emaitza finala lortzeko. Oraingoz C-z idatzitakoa da soilik funtzionalitate osoa daukana, baina helburua da biak guztiz berdinak izatea, ezaugarriei dagokienez.

Zig lengoaiak, gainera, ez du C-ren liburutegi estandarrekiko (*libc*, adibidez) menpekotasunik. Aldiz, bere liburutegi estandar propioa du. Gainera, LLVM erabiltzen du *back-end* moduan, bai C-z bai Zig-ez idatzitako *front-end*-etarako.

1.3 Aurrekariak

Konpiladore batek, oro har, bi atal nagusi ditu: *front-end*-a, eta *back-end*-a. Lehenengoak iturburu kodea irakurtzen du, eta bitarteko kode batera bihurtzen du. Bihurketa honetan iturburu kodearen abstrakzioak sinplifikatzen dira, mihiztadura lengoaiaren egitura antzekoak lortu arte. *Front-end*-ak ez du helburu makinaren ezagutzarik behar. *Back-end*-ak, aldiz, bitarteko kode hori helburu makinaren mihiztadura lengoaiara edo zuzenean makina lengoaiara (nahiz eta gaur egun oso ohikoa ez izan) bihurtzen du.

Banaketa horri esker, konpiladorea beste plataforma baterako konpilatu ahal izateko, *back-end* berri bat besterik ez da idatzi behar. Era berean, iturburu lengoaia desberdin bat konpilatzeko, *front-end* berri bat idaztea nahikoa da.

Back-end ahalmentsu bat erabiliz, era askotako lengoaietarako sortu daitezke konpiladoreak, lengoaia bakoitzarentzat *front-end* bana soilik sortuz. Gainera, *back-end* bat sortzea oso lan zaila da. Izan ere, optimizazio nagusiak bertan gertatzen dira, helburu-makinaren informazio guztia erabili daitekelako.

Horregatik, proiektu honen helburua programazio lengoaia bat diseinatzea, eta horretarako *front-end* bat idaztea izango da. Hortaz, existitzen den *back-end* bat erabiliko da.

Era berean, lengoaia bat analizatzeko, analizatzaile lexiko eta sintaktikoak sortu behar dira. Horretarako, egin beharreko lehenengo ataza lengoaiaren gramatika definitzea da. Honekin, analizatzaile lexiko eta sintaktikoak sortu daitezke. Hala ere, analizatzaile hauek eskuz sortzea lana ekartzen du, eta gramatika aldatzen den bakoitzean, berregin behar dira. Horregatik, hasieran gutxienez, analizatzaile horiek sortzen dituen tresna bat erabiliko da.

1.3.1 *Back-end*-aren hautaketa

Gaur egun bi dira *back-end* nagusiak: GCC eta LLVM. Bi hauek, jatorriz, C lengoaia konpilatzeko sortu ziren, eta GCC eta Clang konpiladoreetan erabiltzen dira nagusiki. Hala ere, bi *back-end* hauek desberdintasun asko dituzte.

Alde batetik, GCC 1987an sortu zen, eta C lengoaian idatzita dago. Hasieran C soilik konpilatzen zuen, nahiz eta aurrerago C++, Fortran eta Ada konpilatzeko zabaldu zen. Linux munduan konpiladorerik erabiliena da, eta sistema eragile honetan instalatuta etorri ohi da. GCC-k bitarteko errepresentazio anitz ditu, ezagunenak GENERIC eta GIMPLE direlarik. Bitarteko errepresentazio hauek oso nahasgarriak dira, eta ez dute oso API garbia zuzenean C++-etik idatzi ahal izateko.

Beste aldetik, LLVM 2003an sortu zen, eta C++ lengoaian idatzita dago. *Back-end* bat ez ezik, hainbat erreminta ditu programak optimizatzeko. Nahiz eta jatorrian C eta C++ konpilatzeko sortu zen, bere diseinua lengoaia hauetatik independentea egin zen. Jatorrian, LLVM GCC-ren *back-end* berri bat bezala sortu zen, baina bere abantailak ikusita, *front-end* berri bat sortzea ere erabaki zuten, eta 2007an Clang sortu zen. Honela, GCC-ren dependentzia guztiz ezabatzea lortu zen, C eta C++ konpilatzen zuen konpiladore oso bat sortuz.

Bi *back-end* hauek ikertu ondoren, proiektu honetarako LLVM erabiltzea erabaki dut, ondorengo arrazoiengatik.

Arrazoi nagusia *front-end*-a idazteko erraztasuna da. Izan ere, GCC-ren kodea ez zen *front-end* berriak idaztea erraza izateko hasieratik diseinatu. LLVM-k aldiz, helburu nagusien artean lengoaia berrietara zabaltzea ahalik eta errazen izatea du.

Gainera, LLVM C++-en idatzita dago, hortaz bere API-a ere. Hasieratik hartu nuen konpiladorea C++-ez idazteko erabakia. Izan ere, behe-mailako kudeaketa ahalbidetzen duen oso lengoaia interesgarria da. Eta nahiz eta graduan zehar erabili dudan, ez dut C++-ez proiektu handirik egin, eta lengoaia sakonago ezagutu nahi dut.

Azkenik, LLVM-ren bitarteko kodea (*LLVM IR* deitutakoa) GCC-rena baino aberatsagoa da. Honi esker, *front-end*-a sinplifikatu daiteke. Gainera, LLVM-k bitarteko kode hau erraztasunez asko optimizatzeko ahalmena du.

1.3.2 Analizatzaile lexiko eta sintaktikoa

Konpilazioa ikasgaian, *yacc*-en ondorengoa den *bison* erabili genuen. Tresna hau oso azkarra da, eta ia Linux sistema guztietan dator instalatuta. Hala ere, bison-ek onartzen duen gramatika modu oso esplizituan idatzi behar da. LALR (*Look-Ahead LR parser*) motako analizatzailea sortzen du, eta analizatzailea erabiltzen duen kodea gramatikan beran sartu behar da.

Analizatzaileak sortzeko beste tresna bat ANTLR (*ANother Tool for Language Recognition*) da. Bere azkeneko bertsioak, ANTLR4-k, ALL(*) izeneko analizatzaile sintaktikoak sortzen ditu. Analizatzaile hauek, Parr 2014-en deskribatutakoak, LL analizatzaileen moldaketa bat dira. Alde batetik, LL(1) analizatzaileen mugak kentzen dira, behar diren *look-ahead* guztiak erabiltzea ahalbidetuz. Gainera, nahiz eta LL analizatzaile baten antzekoa sortzen den, behetik gorako analizatzaileen abantaila batzuk ditu. Izan ere, ALL(*)-ren abantaila nagusia gramatikaren analisia estatikoki (hau da, analizatzailea bera sortzerakoan) egin beharrean, analisia bera egiten den momentuan egitetik dator (hau da, fitxategi

bat analizatzean). Izan ere, honi esker ez dira sarrera posible guztiak kontsideratu behar, soilik sarreran aurkitzen direnak baizik. Honi esker, analisia denbora linealean egitea lortzen du praktikan, nahiz eta ALL(*) analizatzaileak teorikoki $O(n^4)$ ordenakoak diren. Azkenik, ANTLR4 oso bateratuta dago hainbat IDE-ekin, testu bat zuzenean analizatzeko, eta sortutako sintaxi zuhaitz abstraktua ikusteko ahalmena eskainiz.

Arrazoi hauengatik, ANTLR4 erabiltzea erabaki dut. Izan ere, gramatikak oso garbi gelditzen dira, eta asko sinplifikatzen da hauek idaztea.

1.4 Helburua

Proiektu honen helburua *Fleet* izeneko programazio lengoaia diseinatzea, eta lengoaia horretan idatzitako programak konpilatu eta exekutatu ahal izatea da.

Horretarako, Fleet-ren helburuak finkatu behar dira. Oinarritzat C eta C++ lengoaiak hartuko dira. Izan ere, bere helburuetako bat lengoaia eraginkor bat sortzea da. Hala ere, helburua ez da C-ren kopia bat egitea. Izan ere, C-k ez du lengoaia modernoek eduki ditzaketen ezaugarri batzuk.

Ezaugarri moderno horietako batzuen adibideak ondorengoak dira: berezko erroreen tratamendua, *null* balioa duten erakusleak erabili ordez, nahitaez hutsak ez direla ziurtatu behar diren datu motak eskaintzea, etab.

1.5 Irismena

Proiektu hau gauzatzeko denbora mugatua dago, eta programazio lengoaia bat diseinatzea, eta konpiladore bat idaztea lan luzeak dira. Horregatik, nahiz eta proiektua amaitu ondoren garatzen jarraituko den, honen irismena zein den finkatu behar da.

Honetarako, bai lengoaiak bai konpiladoreak eduki behar dituen ezaugarri minimoak eta gehigarriak zehaztuko dira. Honela, proiektua amaitzean ezaugarri minimoak bete direla bermatu beharko da, eta horiek amaitzean geratzen den denboran, ahal diren ezaugarri gehigarri gehienak garatuko dira.

1.5.1 Ezaugarri minimoak

Ezaugarri minimoen artean, edozein lengoaia inperatibotan aurkitu daitezkeen ondorengo egiturak posible izan behar dira:

- **Eragiketa aritmetikoak:** Batuketa, kenketa, biderketa, zatiketa eta hondarra eragiketak oinarrizkoak dira. Gainera, behe mailako kudeaketa ahalbidetzen duten lengoaietan erabilgarriak dira bitak ezkerrera eta eskuinera mugitzeko eragiketak.
- **Eragiketa logikoak:** Bitez biteko eta boolearren arteko *and*, *or*, *not* eta *xor* eragiketak.
- **Erakusleak:** Memoria atzipen zuzena inplementatzeko modurik sinpleena da, eta programatzaileari ahalmen handiagoa ematen dio, egin daitezkeen optimizazioak gauzatzeko.
- **Funtzioak sortu eta deitzeko ahalmena:** Kodea berrerabiltzeko modurik errazena funtzioak sortzea da. Hortaz, kodea hobeto antolatzeko eta berrerabiltzeko, funtzioak sortu eta deitzea ezinbestekoa da.
- **Errekurtsibitatea:** Funtzio-deiak inplementatzean, funtzio errekurtsiboak ahalbidetzea oso erabilgarria izan daiteke.
- **Kontrol egiturak:** Lengoaia inperatiboetan, errepikapenak eta baldintzak erabili ahal izateko, ondorengo egiturak oinarrizkoak dira: `if-else` eta `while`.
- **Datu-motak:** Gutxienez datu-mota bat erabili ahal izango da: zenbaki osokoak. Datu-mota honen bit-zabalera plataformaren araberakoa izango da (adibidez, 32 biteko CPU eta sistema eragileetan, 32 bitekoa izango da).
- **C-rekiko bateragarritasuna:** Programa funtzionatzen duen ala ez jakiteko, erabilgarria izan daiteke pantailan balioak inprimatu ahal izatea. Horretarako, Fleet C-rekin bateragarria egitea izan daiteke sinpleena. Honela, Fleet programa batetik parametroa pantailan inprimatzen duen C funtzio bati dei egin ahal izango zaio.
- **Linux-en konpilatzea:** Fleet lengoaiak ez du ez helburu makinarekiko ez helburu sistema eragilearekiko menpekotasunik. Hala ere, konpiladorea bera nonbait konpilatu behar da. Ondorioz, konpiladorea Linux-en 64 biteko Intel prozesadore batean konpilatzea ezinbestekoa da. Era berean, konpiladoreak daraman Fleet kodea (liburutegi estandar kontsideratu daitekena) plataforma berean konpilatu beharko du (Zehazki, LLVM-ren `x86_64-pc-linux-gnu` helbururako funtzionatu behar du).

1.5.2 Ezaugarri gehigarriak

Ezaugarri minimoak ez bezala, gehigarriek ordena bat daukate. Izan ere, ondorengo ezaugarriak beren lehentasunaren eta behar duten denboraren arabera gauzatuko dira.

1. **Datu-motak:** Behe mailako kudeaketa ahalbidetzen duen lengoaia izanda, tamaina desberdineko zenbaki osokoak, koma higikorreko datuak eta mota boolearrak erabiltzea nahitaezkoa da. Hortaz, 8, 16, 32 eta 64 biteko osokoak, IEEE 754-k estandarrak deskribatutako zehaztasun bakun eta bikoitzeko koma higikorreko zenbakiak eta bi balioko (egia, gezurra) datu-motak inplementatzea ezinbestekoa da.

2. **Oharrak:** Oharrak (iruzkinekin nahasi behar ez direnak) funtzio edo aldagaietan gehitu daitezkeen meta-datuak dira, Java-ren *annotation*-en antzekoak. Datu hauek konpiladoreak edota programak berak erabili ahal izango ditu. Adibidez, funtzioen gainkarga (izen bereko, baina parametro mota edota zenbaki desberdineko funtzioak) inplementatzeko, funtzioen izenak aldatu behar dira; honek funtzio hauek C-rekin bateraezinak bihurtzen ditu. C-rekin bateragarriak izan behar diren funtzioak bereizteko, ohar bat erabili daiteke (`@foreign`).

3. **Aldagai globalak:** Nahiz eta orokorrean funtzioek egoera edo aldagai globalak aldatzea ez den gomendagarria, kasu batzuetan oso erabilgarria izan daiteke. Horregatik, aldagai globalak erabilgarriak izan daitezke. Horiekin batera, C-k dituen *static* aldagaiak inplementatu daitezke, aldagai globalen parekoak direnak, baina funtzio batek bakarrik atzitu ditzaketenak.

4. **Datu-motak sortzeko ahalmena:** Fleet-en helburua Objektuei Orientatutako Programazioa ahalbidetzea da. Hala ere, litekeena da proiektu honek duen denbora murriztapenen ondorioz, Java-k edo C++-ek dituen klaseen antzekoak inplementatzea ezinezkoa izatea. Nolanahi ere, C-ren `struct`-en antzekoak inplementatu daitezke. Hauek oinarrizko datu-motetan oinarrituz, datu-mota konplexuak deskribatzea ahalbidetuko luke.

5. **Exekutagarrien menpekotasunak kentzea:** Hasiera batean, C-rekiko bateragarritasuna aprobetxatuz, programa baten irteera lortzeko, C-z idatzitako eta pantailan inprimatzen duten funtzioak deitu daitezke. Hala ere, ez da oso dotorea programazio lengoaia batek beste batekiko menpekotasuna izatea. Horregatik, ezaugarri honen garapenak C-rekiko *menpekotasuna* ezabatzea izango du helburu. Honela, emaitzak pantailaratzeko, fitxategiak irekitzeko, eta behe mailako beste eragiketa batzuk garatzeko, sistema eragileak berak eskaintzen duen API-a erabiliko du. Linux-en, mihiztadura lengoaia erabiltzea eskatzen du.

6. **Moduluak:** Programazio lengoaia desberdinek modularizazioa era desberdinetan kudeatu dute. Java-k *paketeak* ditu, C++-ek *namespace*-ak. Fleet-ek *moduluak* izan-

go ditu. Hauek Java-ren paketeen antzekoak izango dira. Karpeta berean dauden fitxategi guztiak batera konpilatuko dira. Honela, ez dira fitxategi horietako funtzioak erazagutu behar. Hortaz gain, karpeta desberdinetako fitxategien arteko aldagaien izenak berdinak izatea ahalbidetzen du.

7. **Self-hosted:** C eta C++ lengoaiekiko menpekotasuna are gehiago kentzeko, behin konpiladoreak ahalmen nahikoa edukita, beste *front-end* bat idatzi daiteke, Fleet-ez. Honela, lehenengo C++-ez idatzitako Fleet konpiladorea C++-en konpiladore batekin konpilatuko litzateke. Ondoren, sortutako konpiladore horrekin, Fleet-ez idatzitako Fleet konpiladorea konpilatu daiteke. Azkenik, azken konpiladore horrekin bere burua berriro konpilatu genezake. Prozesu honi *bootstrapping* deitzen zaio, eta konpiladorean inplementatutako optimizazioak konpiladoreak berak erabiltzea ahalbidetzen du.

8. **Analizatzaile sintaktiko propioa:** Behin lengoaia baten gramatika definituta, lengoaia hori aztertzen duen analizatzailea nola sortzen den Konpilazioa ikasgaian ikasi genuen. Hala ere, gramatikan aldaketa txiki batek aldaketa handia ekar dezake analizatzailean. Horregatik, hasieran analizatzaileak sortzen dituen liburutegi bat erabiliko da. Nolanahi ere, analizatzaile horiek, orokorrean, ez dira oso eraginkorrak abiadurari dagokionez. Hortaz, behin gramatika finkatuta, analizatzaile sintaktikoa eskuz idazteak abantailak ekar ditzake.

9. **Portabilitatea:** Nahiz eta gutxienez x86_64 motako prozesadore batean konpilatu behar den, programazio lengoaiaren ahalmena erakusteko, interesrgarria gerta daiteke beste prozesadore ala sistema eragile baterako ere konpilatzea. LLVM oso back-end zabala da, eta plataforma anitzetarako konpilatzea ahalbidetzen du. Baina C-ren liburutegi estandarrarekiko menpekotasuna ezabatuz gero, oinarrizko eragiketa asko (zenbakiak pantailaratzea, adibidez) eskuz inplementatu behar izango dira, plataforma bakoitzerako.

   Hala eta guztiz ere, Fleet-en liburutegi estandarrak interfaze berdina mantendu behar du, helburu-plataforma guztietan zehar. Horretarako, C moduko lengoaiek kodea aurreprozesatzen duten tresnak izan ohi dituzte. Hauen bitartez, kode atal batzuk konpilatzeko edo baztertzeko aukera eskaintzen da, uneko helburu-plataformarako kodea soilik mantenduz. Hala ere, tresna hauek oso sinpleak dira, eta ez dute lengoaiaren ezagutza handia, espero ez diren arazoak sortuz.

   Hau saihesteko, Fleet-ek aurreprozesu bat izan ordez, baldintzazko konpilazioa fitxategi-mailan egingo du. Hau da, plataforma bakoitzerako fitxategi desberdin bat

sortuz, eta atzizki bat gehituz fitxategiaren izenean, konpiladoreak erabaki dezake zein fitxategi behar diren plataforma bakoitzerako. Adibidez, plataformaren izena gehitu daiteke luzapenaren aurretik, puntu batez bananduta. Honela, ‘`builtin.fl`’ fitxategia edozein plataformarako konpilatuko da, baina ‘`builtin.arm.fl`’ eta ‘`builtin.x86_64.fl`’ fitxategiak soilik helburu plataforma ARM eta x86_64 badira konpilatuko dira, hurrenez hurren.

2. KAPITULUA

Diseinua

Programazio lengoaia bat diseinatzeko hainbat urrats jarraitu behar dira. Hasieran, lengoaiaren helburua zein den finkatu behar da. Hau aurreko ataletan egin dugu, baina programazio lengoaia mota desberdin asko daude, hortaz termino teknikoak erabiliko ditugu Fleet-en helburu eta ezaugarriak formalizatzeko.

2.1 Lengoaiaren ezaugarriak

Programazio lengoaia bat diseinatzeko orduan, gauza asko eduki behar dira kontuan. Izan ere, lengoaia konpilatua bada, lengoaia interpretatu baten desberdina izango da. Era berean, goi-mailako lengoaia batek ez du zertan behe-mailako memoria atzipenik eduki, edo erakusleak erabili behar. Horregatik, Fleet diseinatzerako orduan, bere helburua zein den finkatu behar da.

Formalki, Fleet helburu orokorreko, behe-mailako kudeaketa ahalbidetzen duen, sistemak programatzeko, programazio lengoaia konpilatu inperatiboa izango da. Aztertu dezagun kontzeptu horietako bakoitza.

Helburu orokorrekoa Programazio lengoaia bat helburu orokorrekoa izateak domeinu desberdinetan erabilgarria izatea esan nahi du. Honen kontrakoa Domeinu Espezifikoko Lengoaiak (ingelesez *Domain Specific Languages* edo DSL) dira, adibidez Wikipediako artikuluak idazteko erabiltzen den MediaWiki lengoaia. Helburu orokorreko lengoaia bat kontsola-aplikazioak, web zerbitzariak, interfaze grafikoak, bideo-jokoak etab. programatzeko balio behar du.

Behe mailako kudeaketa Definizioz, behe mailako lengoaiak zuzenean makina-kodera itzultzen direnak dira, hau da, mihiztadura lengoaiak. Hala ere, mihiztadura lengoaiak ez diren beste askok ere behe mailako sistemaren kudeaketa ahalbidetzen dute. Adibidez, C-n zuzenean memoria helbideak atzitu daitezke, sistema eragilearekin komunikazio zuzena ahalbidetzen da, eta mihiztadura lengoaia zuzenean C-ren iturburu kodean sartzeko aukera dago.

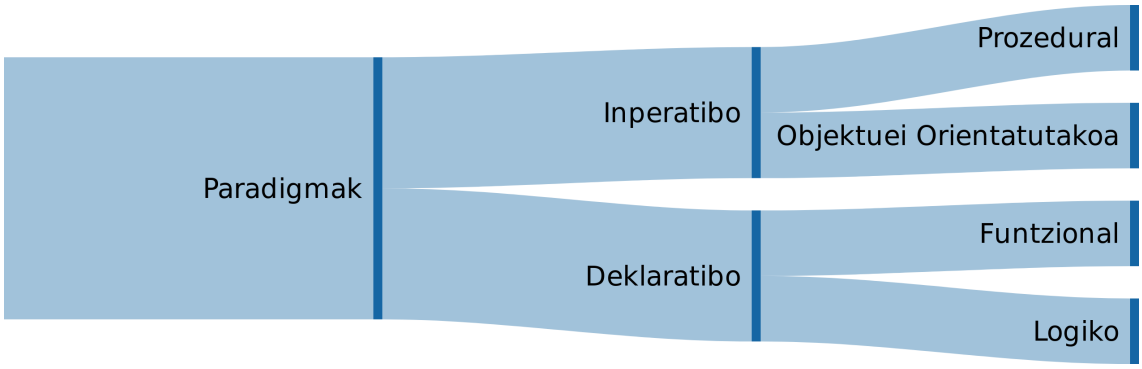
Beste muturrean Python edo Java moduko lengoaiak ditugu. Hauek ez dute memoria helbide bat zuzenean atzitzeko aukera ematen, eta orokorrean beren liburutegi estandarra, edo C-z idatzitako luzapenak erabili behar dira sistema eragilearekin komunikatu ahal izateko.

Fleet-ek behe mailako kudeaketa ahalbidetuko du. Izan ere, ezaugarri hau ezinbestekoa da sistemak programatzeko lengoaia kontsideratu ahal izateko.

Sistemak programatzekoa Programazio lengoaia bat sistemak programatzekoa dela kontsideratzen da sistema-softwarea sortzea ahalbidetzen badu. Honek eskatzen du lengoaiak behe mailako baliabideak kudeatzeko gaitasuna izatea. Adibidez, C, C++, Rust, Zig, D eta beste asko dira sailkapen honetan sartzen diren lengoaiak. Aldiz, Java, Python, PHP eta antzeko lengoaiak ez dira gai zuzenean hardwarearekin komunikatzeko. Sistemak programatzeko lengoaiek, adibidez, gailuetarako kontroladoreak edo sistema eragileak sortu ditzakete.

Konpilatua Oro har, programazio lengoaiak bi motatakoak dira: Interpretatuak ala Konpilatuak. Badira tarteko lengoaiak (Java, adibidez), baina horiek interpretatuen artean kontsideratu daitezke. Izan ere, lengoaia interpretatuek *interprete* baten beharra dute. Honek lengoaia batean idatzitako kodea exekutatu bitartean bihurtzen dute makina kodera. Aldiz, lengoaia konpilatuetan kodea exekutatu baino lehen makina kodera bihurtu behar da, eta hortik aurrera zuzenean exekutatu daiteke. Lengoaia interpretatuek hainbat abantaila dituzte, nagusiena malgutasun handiagoa eskaintzen dutela da. Lengoaia konpilatuak zorrotzagoak izan ohi dira, baina horren truke, eragiketak itzultzen ibili behar ez direnez, exekutatzeko orduan azkarragoak izan ohi dira.

Inperatiboa 2.1 irudian ikusi daitekeen bezala, lengoaia inperatiboak bi paradigma nagusietako batekoak dira. Paradigma inperatiboan, ordenagailuak egin beharreko aginduak lerroz lerro idatzi eta exekutatzen dira. Paradigma honen barruan bi azpi-paradigma nagusi daude: prozedurala, non aginduak prozeduretan multzokatu daitezke, kodea ordenatzea eta berrerabiltzea ahalbidetuz, eta objektuei orientatutakoa, non *objektu*en kontzeptua de-



2.1 Irudia: Programazio paradigmak erakusten dituen diagrama.

finitzen den, datuak eta horiek manipulatzen duten kodea batera errepresentatzeko aukera eskaintzen duen. Fleet, hasieran prozedurala besterik ez da izango, baina objektuetara orientatzeko helburua du, C++ edo Python-en antzera. Hau da, klase baten barnean ez dauden prozedura solteak sortu ahal izango dira.

Beste paradigma nagusia, deklaratiboa, programak egin beharrekoa *deklaratzen* du (hortik bere izena), baina ez du esaten zein ordenean egin behar den ataza bakoitza. Horren ordez, lengoaiaren inguruneak beharrezko pausoak hartu behar ditu.

2.2 Inplementatutako funtzionalitatea

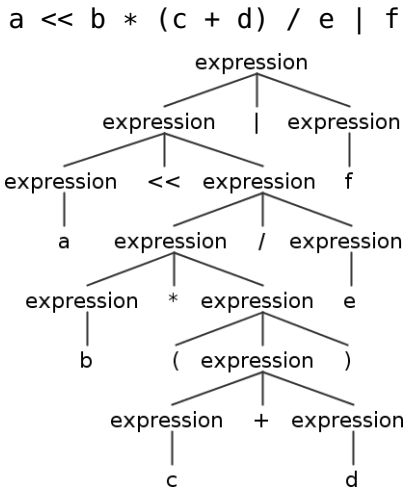
Irismenaren atalean aipatu bezala, programazio lengoaia bat sortzea lan luzea da. Izan ere, C++, Java, Python, R, Javascript eta PHP gaur egun gehien erabiltzen diren lengoaien artean daude. Horiek guztiek 20 urte baino gehiago dituzte, eta oraindik bertsio berriak kaleratzen dituzte erregularki. Hortaz, atal honetan proiektu honetarako finkatu diren ezaugarri minimoak, eta gauzatzea lortu diren ezaugarri gehigarriak azalduko dira, adibideekin batera. Honela, Fleet programazio lengoaiarekin egin daitekena ulertu daiteke.

Eragiketa aritmetiko, logiko eta boolearrak Oinarrizko eragiketa aritmetikoak (batuketa, kenketa, biderketa, zatiketa eta hondarra) inplementatu dira. Horiekin batera, bitak maneiatzeko beste eragiketa batzuk ere inplementatu dira, besteak beste biten desplazamenduak, bai eta bitez biteko ‘*and*’, ‘*or*’, ‘*xor*’ eta ‘*not*’ eragiketak ere.

Ondorengo zerrendan ikusi daiteken bezala, eragiketa hauek lehentasun baten arabera daude ordenatuta. Lehenengo agertzen direnak ondoren daudenak baino lehentasun handiagoa dute. Kontuan izan adierazpenak multzokatu daitezkela parentesiak erabiliz, nahi den eragiketa-ordena lortzeko.

1. Biderketa, zatiketa eta hondarra (`*`, `/` eta `%`)
2. Batuketa eta kenketa (`+` eta `-`)
3. Bit desplazamenduak (`>>>`, `>>` eta `<<`)
4. Konparaketa eragileak (`<=`, `>=`, `<` eta `>`)
5. Berdinketa eta desberdinketa (`==`, `!=`)
6. Bit mailako *and* (`&`)
7. Bit mailako *xor* (`^`)
8. Bit mailako *or* (`|`)
9. *And* boolearra (`&&`)
10. *Or* boolearra (`||`)

Adibidez, ondorengo adierazpena izanda, azpian ikusi daiteken sintaxi zuhaitza sortuko litzateke.



Fleet konpiladoreak honela itzuliko luke aurreko adierazpena LLVM IR-ra:

```
%add = add i64 %c, %d
%mul = mul i64 %b, %add
%div = sdiv i64 %mul, %e
%shl = shl i64 %a, %div
%bit_or = or i64 %shl, %f
```

Kontuan izan behar da eragile boolearrek ez dutela zirkuitulaburrik egiten. Hortaz, C-z `if (current != 0 && (*current).size > 0)` idatzi daiteken bitartean, Fleet-ez bi `if` sententzietan banatu behar da.

Erakusleak Erakusleak helbide bat gordetzen duten datu-motak dira. Erakusle izena balio bat seinalatzen edo erakusten dutelako da. Fleet-en, C-ren antzera, erakusleak datu-mota batekoak izan behar dira, baina adierazpena alderantziz idazten da. Hau da, `int` datu mota izanda, Fleet-en, `*int` izango da mota horri seinalatzen duen helbide baten datu-mota. Aldaketa honen arrazoia ingelesez irakurtzen den modua da. Izan ere, aurreko datu mota hori ingelesez *«pointer to int»* irakurtzen da, Fleet-en idazten den ordena berdinean.

Kontrol egiturak Oinarrizkoak diren `if-else` eta `while` egiturez gain, `do-while` eta `for` egiturak inplementatu dira. Lehenengoa bitarteko kodera bihurtzeko oso sinplea delako, eta bigarrena erabilgarria delako, adibidez, taulak korritzeko. C-n bezala, hiru atalak hautazkoak dira. Hortaz, '`while(true){...}`' eta '`for(;;){...}`' baliokideak dira. Kontuan izan `break` eta `continue` motako sententziak ez daudela inplementatuta. Ondorioz ezinezkoa izango da begizta horietatik irtetea, `return` aginduak erabili gabe.

Adibidez, ondorengo programan ikusi daiteke `for` sententzia nola erabili daiteken.

```
int main(int argc, **i8 argv) {
    for (int i = 0; i<10; i++) {
        println(fib(i));
    }
    return 0;
}
```

Aurreko funtzio hori LLVM IR-ra bihurtzeko, kodea oinarrizko blokeetan[1] banatu behar da, ondoren ikusi daiteken bezala.

```
define internal i64 @main_i64PPi8(i64 %argc, i8** %argv) {
main_entry:
    %main_i64PPi8_i = alloca i64
    store i64 0, i64* %main_i64PPi8_i
    br label %forcond
forcond:
    %i = load i64, i64* %main_i64PPi8_i
    %lt = icmp slt i64 %i, 10
    br i1 %lt, label %for, label %forcont
for:
    %i1 = load i64, i64* %main_i64PPi8_i
    %fib = call i64 @fib_i64(i64 %i1)
    call void @println_i64(i64 %fib)
```

[1]Oinarrizko blokea sarrea eta irteera puntu bakarra duen agindu segida bat da. Hau da, programaren beste atal batetik soilik blokearen lehen agindura egin daiteke jauzi, eta blokearen azken bloketik bakarrik egin daiteke jauzi.

```
    %i3 = load i64, i64* %main_i64PPi8_i  ; Eguneraketa
    %add = add i64 %i3, 1
    store i64 %add, i64* %main_i64PPi8_i
    br label %forcond
forcont:
    ret i64 0
}
```

Funtzioak eta errekurtsibitatea Fleet-en funtzioak sortu eta deitu daitezke, bai eta bere burua deitzen duen funtzio errekurtsiboak sortu ere. Honi esker, berrerabili daitezkeen eta izen esanguratsua duten kode zatietan antolatu daiteke kodea. Gainera, grafoen sakonerako korritzea, Euklides-en algoritmoa eta Fibonacci segida (azpian ikusi daitekena) bezala, naturalki errekurtsiboak diren algoritmoak erraz inplementatzea ahalbidetzen du.

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

Bere itzulpena LLVM IR-ra honakoa da.

```
define internal i64 @fib_i64(i64 %n) {
fib_entry:
    %le = icmp sle i64 %n, 1
    br i1 %le, label %then, label %else
then:
    ret i64 1
else:
    %sub = sub i64 %n, 1
    %fib = call i64 @fib_i64(i64 %sub)
    %sub1 = sub i64 %n, 2
    %fib2 = call i64 @fib_i64(i64 %sub1)
    %add = add i64 %fib, %fib2
    ret i64 %add
}
```

Nahiz eta aurreko adibideetan ikusi daiteken, ondorengo adibidean funtzioen gainkarga nola inplementatu den ikusi daiteke. Bi funtzio daude ‘`f`’ izenekoak, baina batek osoko motako parametro bat hartzen du, eta besteak balio boolear bat. Honela, ‘`f`’ funtzio baten deia ikustean, bere parametroen arabera deitu behar den funtzioa erabakiko da.

```
void f(int a) {
```

```
    println("Osoko bat hartzen dut");
}
void f(bool a) {
    println("Boolear bat hartzen dut");
}
int main(int argc, **i8 argv) {
    f(0);
    f(true);
    return 0;
}
```

LLVM IR-ra bihurtuz, ikusi daiteke bi funtzioek izen desberdina dutela, bere parametroen araberakoa. Izan ere, LLVM IR-n ezin dira funtzioak gainkargatu, hortaz funtzioen izena aldatu behar da.

```
define internal i64 @main_i64PPi8(i64 %argc, i8** %argv) {
main_entry:
  call void @f_i64(i64 0)
  call void @f_b(i1 true)
  ret i64 0
}
define internal void @f_i64(i64 %a) {
f_entry:
  call void @println_Pi8(i8* getelementptr inbounds ([22 x i8], [22 x i8]* @str
      ↪ .2, i64 0, i64 0))
  ret void
}
define internal void @f_b(i1 %a) {
f_entry:
  call void @println_Pi8(i8* getelementptr inbounds ([24 x i8], [24 x i8]* @str,
      ↪  i64 0, i64 0))
  ret void
}
```

Datu-motak Datu-motak bi multzotan banatzen dira. Alde batetik, lengoaiak definitzen dituen motak daude (berezkoak), eta beste aldetik, erabiltzaileak Fleet kodea idatziz definitzen dituenak (sortutakoak).

Aldi berean, berezko motak bi multzotan banatu daitezke, oinarrizkoak eta eratorritakoak. Oinarrizkoak 16 dira:

- **`void`:** Daturik eza errepresentatzen du. Datu-mota hau soilik funtzioen emaitza moduan erabili daiteke, prozedura bat dela, hau da, emaitzarik ez duela esateko.

- **`bool`:** Datu-mota boolearra. Bi balio izan ditzake `true` (egiazkoa) eta `false` (faltsua). Baldintza bat espero denean, mota honetako datu bat soilik onartuko da. Hortaz, Fleet-en ezin da idatzi C-z ohikoa den '`while(1)`', eta horren ordez '`while(true)`' idatzi beharko da.

- **`float` eta `double`:** Zenbaki arrazionalak errepresentatzeko datu-motak dira. IEEE 754 estandarrak definitutako zehaztasun bakuneko eta bikoitzeko koma higikorreko balioak gordetzen dituzte, hurrenez hurren.

- **`i8`, `i16`, `i32`, `i64`, `i128`, `u8`, `u16`, `u32`, `u64` eta `u128`:** Zenbaki osokoak errepresentatzeko balio dute. `i` letraz hasten direnak zeinudunak dira, hau da, zenbaki positiboak eta negatiboak gorde ditzakete. `u` letraz hasten direnak, aldiz, zeinugabeak dira, hau da, zenbaki positiboak soilik gorde ditzakete. Letraren ondorengo balioa datuak okupatuko duen bit-zabalera da. Mota zeinugabeko aldagaiek $[0, 2^w - 1]$ tarteko balioak gorde ditzakete, non 'w' motaren bit-zabalera den. Mota zeinudunek balioak biko osagarrian gordetzen dituzte. Ondorioz, mota horietako balioek $\left[-2^{w-1}, 2^{w-1} - 1\right]$ tarteko balioak gordetzen dituzte.

- **`int` eta `uint`:** Datu-mota hauek bereziak dira. Izan ere, aurreko mota guztiak berdin berdinak dira plataforma guztietan. Hauek, aldiz, nahiz eta `iX` eta `uX` motako motak bezala zenbaki osokoak gordetzen dituzten, plataformaren arabera beren zabalera aldatzen da. Hortaz, 64 biteko ordenagailu batean, `int` eta `uint` motak `i64` eta `u64` moten baliokideak izango dira, eta 32 biteko plataforma batean, `i32` eta `u32` motenak. Hau da, ez da esplizitu aukeratu behar bit zabalera, eta automatiko aukeratzen da CPU-arentzat eraginkorrena dena.

Eratorritako motak, taulak eta erakusleak dira. Izan ere, datu-mota hauek beste datu-mota batean oinarritu behar dira.

Fleet lengoaian ez da datu moten arteko bihurketa inplizitua egiten. Orokorrean, bi datu mota desberdinen artean ezin da inolako eragiketarik egin. Salbuespen bakarra erakusle eta zenbaki osokoen artean da. C-n bezala, erakusleei zenbaki osoko bat gehitu edo kendu ahal zaio, hurrengo edo aurreko elementuen helbidea lortzeko. Datu moten artean bihurtzeko, bihurketa esplizitua egin behar da, ondorengo adibidean ikusi daiteken bezala.

```
1 double batu(double x, int y) {
```

```
    return x + (double) y;
}
```

Datu motak konbinatuz, beste mota berri batzuk sortu daitezke, C-ren *struct*-en parekoak. Ondorengo adibidean, 2 dimentsioko bektore bat errepresentatu dezaken egitura sortzen da, eta horren elementuak atzitzen dira bektorearen luzeraren karratua kalkulatu ahal izateko.

```
class Bektore2D {
    int x;
    int y;
}

int luzerakarratu(Bektore2D b) {
    return b.x * b.x + b.y * b.y;
}
```

C-rekiko bateragarritasuna Fleet-ek gainkarga baimentzen du. Hau da, parametroen kopuru edo motetan soilik desberdinak diren izen berdineko funtzioak sortu daitezke. Hau inplementatzeko, konpiladoreak funtzioen izenak aldatu egiten ditu (*name mangling* ingelesez). C-n, aldiz, ezin da horrelakorik egin. Hortaz, Fleet eta C-ren funtzioak ez dira zuzenean bateragarriak.

Hau konpontzeko, Fleet-ek gainkarga desaktibatzeko eta funtzioek bere jatorrizko izena mantentzeko aukera eskaintzen du. Horretarako funtzioei `@foreign` oharra gehitu behar zaio funtzio bati. Hori funtzio askotan idaztea luzea gertatu daitekenez, funtzio baten izena traol ikurrarekin (#) hasiz gero, emaitza bera lortuko da.

Azkenik, Fleet-ek, lehenespenez, kodea konpilatu eta estekatzen du. C-rekin batera kodea konpilatu ahal izateko, kodea konpilatu besterik ez du egin behar. Ondoren, C konpiladore bat erabiliz, C kodea konpilatu behar da. Azkenik, estekatzaile bat erabiliz, C objektu kodea eta Fleet objektu kodea batera estekatu daitezke, exekutagarria sortzeko.

Linux-en konpilatzea LLVM-ri esker, konpiladorea plataforma batean konpilatu ondoren, honek plataforma desberdinetarako konpilatu dezake Fleet kodea. Hala ere, posible da plataforma bakoitzak ezaugarri batzuk bakarrik onartzea. Adibidez, 32 biteko prozesadoreek ez dituzte 128 biteko erregistrorik. Hortaz, ezin dira tamaina horretako aldagaiekin zuzenean kalkuluak egin. LLVM-k ahalbidetzen du eragiketa hauek software bidez simulatzea, baina hau ez egitea erabaki da, eraginkortasuna asko murrizten delako.

Hala ere, Fleet kodea ARM plataformarako konpilatzea lortu da. Honetarako, Raspberry Pi 3 ordenagailu batean Raspbian sistema eragilea duelarik, Fleet kode konpilatua exekutatu da.

Exekutagarrien menpekotasunak kentzea Nahiz eta Fleet konpiladoreak C++-en liburutegi estandarra behar duen, honek konpilatzen duen Fleet kodeak ez du beste lengoaia batean idatzitako kodearekiko menpekotasunik. Izan ere, Fleet-en helburu makinaren mihiztadura lengoaia sartu daiteke, zuzenean sistema eragilearekin komunikatu ahal izateko (eten bat sortuz, adibidez).

Honek abantaila eta desabantaila batzuk ditu. Alde batetik, lengoaia independente egiten du. Honi esker, lengoaian arazo bat egonez gero, ziur egon gaitezke arazoa gure kodean dagoela, eta ez beste nonbaiten. Beste aldetik, C-k eskaintzen zigun portabilitatea gure esku gelditzen da. Izan ere, pantailan digitu bat inprimatzea bezalako oinarrizko eragiketak plataforma bakoitzerako inplementatu behar dira, eta hau era oso desberdinetan egin behar izan daiteke.

Portabilitatea Programa bat bi plataforma edo gehiagotarako garatzen den bakoitzean, bi zatitan banandu daiteke. Alde batetik, kodea ahal den gehien berrerabili ahal izateko, plataforma guztietarako komuna den kodea dugu. Beste aldetik, helburu plataforma bakoitza desberdina denez, kode zati batzuk plataforma bakoitzarentzat modu desberdinean idatzi behar dira. Hau ahalbidetzeko, Fleet-en fitxategi-mailako baldintzazko konpilazioa inplementatu da. Honela, plataforma guztietarako komuna den kodea espero den moduan idatzi daiteke, eta ondoren, plataforma bakoitzerako berezia dena separatutako eta izenean atzizki bereziak dituzten fitxategietan idatzi daiteke. Honela, konpiladoreak jakin ahal izango du zer kode behar den plataforma bakoitzerako konpilatzeko.

Behe-mailako kudeaketa Fleet lengoaiak hainbat baliabide ditu behe-mailako sistemaren kudeaketa ahalbidetzeko. Alde batetik, erakusleak erabili daitezke memoria zuzenean atzitu ahal izateko. Beste aldetik, Fleet kodean zuzenean helburu makinaren mihiztadura lengoaia txertatzea posible da.

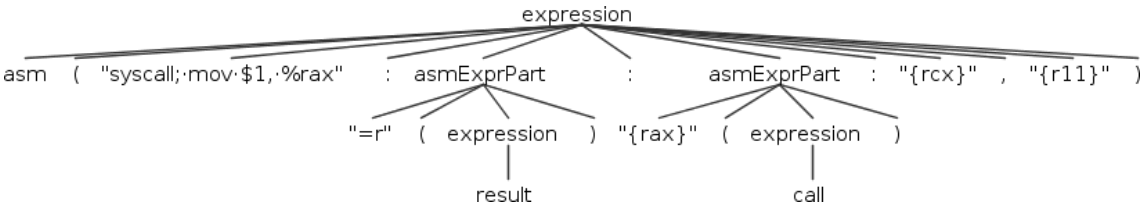
Azken honi esker, Fleet-en liburutegi estandarrak zuzenean balioak pantailaratu ditzake, beste liburutegi estandarrik erabili gabe. Hau gauzatzeko, Linux sistema eragilearen sistema-deiak (*system-call*) erabili dira, bai x86_64 agindu-multzoaren, bai ARM-renaren agindu bereziak erabilita.

Ondorengo adibidean mihiztadura-lengoaia zatiak Fleet-en nola txertatzen diren ikusi dai-

teke. Adibide hau parametrorik ez duen x86_64 plataforman Linux-en sistema-dei bat egiteko balio du (*fork* edo *getuid* deiak, adibidez).

```
1 int syscall(int call) {
2     int result;
3     asm("syscall; mov $1, %rax":"=r"(result) :"{rax}"(call) : "{rcx}", "{r11}");
4     return result;
5 }
```

Kode horren `asm` adierazpenak sortzen duen sintaxi zuhaitz abstraktua ondoren ikusi daiteke.



Azkenik, lengoaiaren ahalmena probatzeko, mihiztadura-lengoaia txertatuz, sistema eragileari memoria dinamikoki eskatzea lortu da, C-ren `malloc` funtzioaren antzeko funtzionamenduarekin (oraindik memoria askatzea, hau da, `free`-ren parekoa, ez da inplementatu). Uneko inplementazioa oso gordina da, eta ez du `malloc` funtzioak duen eragin-kortasuna, ez abiadura aldetik, ez memoria-erabilera aldetik. Baina lehen inplementazio moduan bere helburua betetzen du. Funtzio hau Fleet-en liburutegi estandarrean sartu da. Hortaz, zuzenean erabili daiteke idatzitako programetatik.

3. KAPITULUA

Inplementazioa

Kapitulu honetan konpiladorearen inplementazioa nola gauzatu den deskribatuko da. Hasteko, lengoaia gramatika batekin formalki nola deskribatu den aztertuko da. Ondoren sintaxi zuhaitza osatzen duten klaseak aztertuko dira. Jarraian, zuhaitz hori korrituz, LLVM IR-a nola sortzen den azalduko da. Azkenik, erroreen tratamendua nola egin den azalduko da.

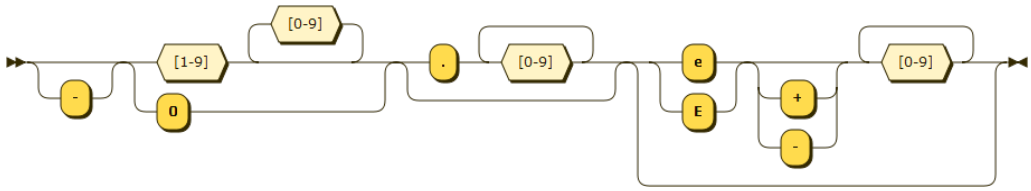
3.1 Lengoaiaren deskribapen formala: gramatika

Fleet-en gramatika ahalik eta sinpleena izango da. Alde batetik, honen aldaketak errazagoak izateko, eta beste aldetik, etorkizunean analizatzaile lexiko eta sintaktikoa eskuz idazterakoan, lan gutxiago egin behar izateko.

Gramatika bat definitzeko, hainbat era desberdin daude, diagrametatik, formalizatutako meta-lengoaiak erabiltzera arte.

Diagramen adibide bat 3.1 irudian ikusi daiteke. Diagrama horiek *Railroad Diagram* edo trenbide diagrama izena dute, trenbideen antzera, ezkerretik eskuinera dauden bide desberdinek lengoaiak onartzen dituen gramatikak definitzen dituelako. Hauetan, lerroak jarraituz, adierazpen erregular batek onar ditzaken sarrerak aztertu daitezke. Honelako hainbat erabilita, lengoaia baten gramatika definitu daiteke. Diagrama hauen abantaila nagusia ulertzeko erraza dela da. Adierazpen erregularrekin ohituta egonez gero, begirada bat nahikoa da diagrama hauetako batek onartzen duen testua ulertzeko.

Diagrama horiek oso erabilgarriak dira gizakiok ulertzeko, baina ordenagailu batek ez du erraztasun hori. Horregatik, programa batek analizatzaile lexiko eta sintaktikoak sortu



3.1 Irudia: JSON lengoaian idatzi daitezkeen zenbakien sintaxia definitzen duen diagrama.

ditzan, beste era batera definitu behar dira gramatikak. Honetarako, *meta-lengoaiak* sortu dira. Hau da, lengoaiak definitzeko erabili daitezkeen lengoaiak berak (eta ondorioz, bere burua definitu dezaketenak).

Arazoa da ez dagoela ondo estandarizatutako meta-lengoaia zabalik. Horregatik, ondorengo paragrafoetan, BNF, EBNF, ABNF eta ANTLR4 meta-lengoaiek dituzten abantailak eta desabantailak deskribatuko ditut.

Alde batetik, estandarizatutako meta-lengoaia bakarra *Backus–Naur Form* edo BNF da. John Backus IBM-ko ingeniariak eta Peter Naur-ek sortu zuten, ALGOL 58 lengoaiaren sintaxia formalizatzeko asmoz. Honen arazoa malgutasun falta da. Izan ere, BNF-z idatzitako gramatikak oso errepikakorrak dira, errepikapenak definitzeko ezin direlako adierazpen erregularretan erabili ohi diren Kleene izarra bezalako eragiketak erabili. Horregatik, beste meta-lengoaia batzuekin konparatuta, erregela gehiago idatzi behar ohi dira.

Hala ere, BNF asko zabaldu zen, eta honetan oinarritutako beste meta-lengoaia asko sortu ziren. Gaur egun gehien ezagutzen direnak EBNF eta ABNF dira.

Alde batetik, ABNF (*Augmented BNF*) IETF-k (interneteko estandarrak definitzen dituen erakundeak) definitzen dituen lengoaiak deskribatzeko erabiltzen da (ikus. Crocker & Overell 2008). Lengoaia honetan, literalak kakotsen artean gehitu daitezke, BNF-k zituen murriztapenak ezabatuz. Gainera, errepikapenak aurrizki bidez laburtu daitezke (‘`4DIGITU`’ katea ‘`DIGITU DIGITU DIGITU DIGITU`’ katearen baliokidea da), eta Kleene izarraren pareko eragiketak ere onartzen ditu.

Beste aldetik, EBNF (*Extended BNF*) meta-lengoaia familia bat dela esan daiteke. Familia honen barnean, BNF-n oinarritutako eta hortik zabaldutako meta-lengoaia asko sartzen dira. Horien artean, batek ISO 14977 estandarra jarraitzen du (ikus. *Information technology – Syntactic metalanguage – Extended BNF* 1996). Baina estandar horretan definitutako EBNF lengoaia ez da oso erabilia. Izan ere, (Jinks 2004) artikuluan aipatzen den bezala, bi EBNF mota nagusi daude: errepikapenak sinplifikatzeko adierazpen erregularretan oinarritzen diren EBNF-ak, eta Niklaus Wirth-ek sortutako WSN meta-lengoaian oinarritutakoak.

Gaur egun, adierazpen erregularrak oso erabiliak dira. Lengoaia ezagunenek beren liburutegi estandarretan dute horiek tratatzeko tresnak. Horregatik, hauetan oinarritutako EBNF lengoaiak dira gehien erabiltzen direnak, eta erabiltzen hasteko errazenak.

Aurretik esan bezala, konpiladoreak erabiliko dituen analizatzaileak sortzeko ANTLR4 erabiliko dut. Tresna honek gramatikak definitzeko bere sintaxi propioa du, adierazpen erregularren antzekoa den EBNF-n oinarritutakoa, eta hori erabili behar da ANTLR4 erabiltzeko.

Gramatika bat definitzen duten fitxategiek '`.g4`' luzapena izan behar dute, eta fitxategiaren izena gramatikaren berdina izango da. Fitxategiaren barnean, lehenengo lerroan gramatikaren izena errepikatu behar da. Honen ondoren, gramatika bera hasten da. Bi erregela mota daude: analisi sintaktikoaren erregelak, eta analisi lexikoarenak. Lehenengoak letra xehez hasi behar dira, eta normalean fitxategiaren hasieran jartzen dira. Bigarrenak letra larriz hasi behar dira, eta erregela sintaktikoen ondoren idatzi ohi dira. Letra xehe eta larriak erabiltzearen abantaila nagusia biak desberdintzeko erraztasuna da.

Erregela bakoitza bere izenaz hasten da, ondoren bi puntu ikurra, eta jarraian dituen produkzioak, marra bertikal batez bananduta. Erregeletan, erregela lexikorik ez dituzten literalak sortu daitezke, kakotxa bakunen artean. Honek lexikoaren partea idaztea sinplifikatzen du, baina ez du ikur horiei izena emateko aukerarik ematen (ondoren inplementazioan identifikatu ahal izateko). Lehen esan bezala, sintaxia adierazpen erregularrenaren antzekoa da, hau da, errepikapenak adierazteko, izar ($*$) eta gehi (+) ikurrak erabiltzen dira, eta aukerako atalak definitzeko, galdera ikurra (?). Atalak multzokatzeko parentesiak erabiltzen dira.

Azkenik, erregelak puntu eta koma ikurrarekin amaitzen dira. Gainera, erregela lexikoek adierazpen erregularretan ohikoak diran karaktere multzoak erabili ditzakete.

Fleet lengoaiaren gramatika A eranskinean ikusi daiteke. Gramatika hori sinplifikatuta dago. Izan ere, inplementazioan erabiltzen den gramatikan produkzio desberdin bakoitzari identifikatzaile bat gehitu zaio, kodetik jakin ahal izateko zein produkzio den analizatzen duguna, baina ez da beharrezkoa lengoaia definitzeko. Gainera, ANTLR-ek zehaztapen berezi batzuk egingo ez balitu, gramatika hori anbiguoa izango litzateke. Zehazki, anbiguotasunaren aurrean, gramatikan definitutako lehenengo produkzioa aukeratuko da. Hortaz, eragiketen lehentasuna zehazteko, lehentasun handienekoak lehenago definitzea nahikoa da.

Behin gramatika edukita, ANTLR-eri fitxategi hori pasa diezaiokegu, C++-en analizatzailea sortzeko esanez. Sortutako analizatzaileak, karaktere kate bat emanda, gramatikak

definitzen duen lengoaiaren barneko katea den ala ez esango digu.

3.2 Fleet-en adierazpena memorian: Sintaxi zuhaitza

Konpiladorearen helburu nagusia Fleet lengoaia LLVM IR-ra (LLVM-k ulertzen duen bitarteko kodera) bihurtzea da. Horretarako, ANTLR-k sortutako analizatzailea erabiliko du. Analizatzaileak, analisi lexiko eta sintaktikoa burutu bitartean, bere sintaxi zuhaitz abstraktu[1] propioa sortzen du. Zuhaitz horren arazoa nagusia analisiarenak diren propietate anitz dituela, eta eskuz zuhaitz hori korritzea eta dekoratzea[2] oso erosoa ez dela da.

Hortaz, zuhaitz hori korrituko dugu, gure sintaxi zuhaitz propioa sortzen dugun bitartean. Zuhaitza korritu ahal izateko, ANTLR-ek bi baliabide sor ditzake: *listener* eta *visitor*. Biak C++ (edo ANTLR-ek onartzen duen beste helburu lengoaia bat) klase bat sortzen dute. Horien azpiklase bat sortzean, zuhaitza korritu ahal izateko erraztasunak eskainiz.

Lehenengo baliabideak, *listener* izenekoa, zuhaitza automatikoki korritzen du, eta gainidatzi daitezkeen funtzio batzuk deitzen ditu produkzio bakoitza korritu aurretik, eta ondoren.

Bigarrenak, *visitor* izenekoa, korritzearen eskeletoa eta funtzio laguntzaile batzuk eskaintzen ditu. Hortaz, garatzailea izan behar da zuhaitza nola korritu erabakitzen duena. Honek malgutasun handiagoa eskaintzen du, azpi-zuhaitz bat behin baino gehiagotan korritzea, edo guztiz baztertzea ahalbidetzen duelako. Gainera, zuhaitza era naturalago batean korritzen da, eta erregela bakoitzak C++-en objektu bat bueltatu dezake. Hortaz, ANTLR-ek sortutako zuhaitza ez denez oso erosoa, baina analitzatzailea erabili nahi denez, bisitaria erabiliko dugu, ANTLR-en zuhaitza gure SZA propiora bihurtzeko (hau '`Visitor.cpp`' fitxategian dago inplementatuta).

Sintaxi zuhaitza osatzen duten klaseak sortzeko, C++-ek eskaintzen duen objektuei orientatutako programazioaz eta bereziki klaseen herentziaz baliatu naiz. Klaseen diagrama B eranskinean ikusi daiteke. Ondoren azaltzen da sintaxi zuhaitz abstraktua osatzen duten klase bakoitzak errepresentatzen duen egitura.

3.2.1 Goi mailako elementuak

- **`Module:`** Konpilazio-unitate bat errepresentatzen du, hau da, batera konpilatzen diren elementuen multzoa. Funtzioen deklarazioak, definizioak eta egituren defini-

[1] Programazio lengoaia batean idatzitako programa baten egitura errepresentetzen duen zuhaitz egitura.

[2] Sintaxi zuhaitz abstraktu baten elementuei analisitik lortutako informazioa gehitzea. Adibidez, adierazpen baten emaitza, edo funtzio baten parametroen motak.

zioak gordetzen ditu.

- **`Annotation:`** Ohar bat errepresentatzen du. Hauek funtzio edo egituretan jarri daitezke, konpilazio- edo exekuzio-garaian informazio moduan erabiltzeko. Adibidez, funtzio batean '`@foreign`' oharra egonez gero, konpiladoreak ez du funtzio horren izena aldatuko. Oraingoz hori da oharren erabilera bakarra, eta ezin da informazio hori exekuzio garaian lortu, baina helburua da hori ahalbidetzea.

- **`FunctionDeclaration:`** Funtzio baten deklarazioa errepresentatzen du. Lau eremu nagusi gordetzen ditu: funtzioaren izena, emaitzaren datu-mota, argumentuen izenak eta motak eta funtzioak izan ditzakeen oharrak.

- **`Function:`** `FunctionDeclaration` klasearen azpiklasea da. Funtzio baten definizioa errepresentatzen du, hau da, deklarazioaz gain, funtzioaren inplenetazioa ere badu. Aurrekoak gordetzen duenarekin batera, barnean dituen sententzien zerrenda gordetzen du.

- **`StructDeclaration:`** Klase edo egitura baten definizioa errepresentatzen du. Bere izena eta eremuen izenak eta datu-motak gordetzen ditu, oharrekin batera. Zeharka errekurtsiboak diren egiturak definitzea ahalbidetzen da. Hau da, egiturek ezin dute bere motako eremurik izan, Horren ordez, bere motari erakuslea den balio bat gorde dezake (hau oso erabilgarria da lista estekatuak inplementatzeko).

3.2.2 Sententziak

- **`VarDeclStmtAST:`** Aldagai baten erazagupena errepresentatzen du. Aldagaiaren izena eta mota gordetzen ditu. Aldagai lokalak ez ezik, aldagai globalak ere klase honen instantzia batekin adierazten dira. Aldagaiaren hasieraketa ere gorde dezake, egonez gero.

- **`AssignStmtAST:`** Esleipen bat errepresentatzen du. Ezkerreko aldagaia, eskuineko adierazpena eta esleipen mota (`=, +=, %=, etab.`) gordetzen ditu.

- **`IfStmtAST:`** `if` sententzia bat errepresentatzen du. Baldintzaren adierazpena, hau egiazkoa izanez gero exekutatu behar den sententzia, eta gezurra izanez gero exekutatu beharrekoa gordetzen ditu.

- **`WhileStmtAST:`** `while` sententzia bat errepresentatzen du. Baldintzaren adierazpena eta errepikatu beharreko sententzia gordetzen ditu.

- **`ReturnStmtAST:`** `return` motako sententziak errepresentatzen ditu. Emaitza izango den adierazpena gordetzen du.
- **`ExprStmtAST:`** Adierazpen bat sententzia moduan errepresentatzeko balio du. Adibidez, funtzio bati dei egitean emaitza ez bada erabiltzen, hau da, lerro batean `print(5);` agertzen bada. Adierazpena gordetzen du.
- **`BlockStmtAST:`** Erazagupenak multzokatu eta antolatzeko, eta baldintza eta begiztetan sententzia bat baino gehiago sartu ahal izateko erabiltzen da. Bere barnean dauden sententziak gordetzen ditu.
- **`ForStmtAST:`** `for` begizta bat errepresentatzen du. Lau eremu gordetzen ditu: hasieraketa, baldintza, eguneraketa, eta barneko sententzia.
- **`DoWhileStmtAST:`** `do-while` begizta bat errepresentatzen du. Gordetzen dituen eremuak baldintza eta barneko sententziak dira.

3.2.3 Adierazpenak

- **`LiteralBoolExprAST:`** Boolear balio literalak (‘`true`’ eta ‘`false`’) gordetzen ditu. Balioa hori gordetzen du.
- **`IntExprAST:`** Zenbaki osoko balio literalak gordetzen ditu. 8, 16, 32, 64 eta 128 biteko zenbakiak errepresentatu ditzake, bai zeinudunak bai zeinugabeak. Gainera, zenbaki horiek oinarri desberdinetan idatzita etor daiteke: bitarrez, hamartarrez eta hamaseitarrez. Balioa karaktere kate moduan, oinarria, tamaina eta zeinuak gordetzen ditu.
- **`FloatExprAST:`** IEEE 754-1985 estandarrak definitutako zehaztasun bakuneko koma higikorreko zenbaki hamartar literalak gorde ditzake. Balioa gordetzen du.
- **`DoubleExprAST:`** IEEE 754-1985 estandarrak definitutako zehaztasun bikoitzeko koma higikorreko zenbaki hamartar literalak gorde ditzake. Balioa gordetzen du.
- **`StringExprAST:`** Karaktere kate literala. C-ren parekoak dira. Fleet iturburu kodea UTF-8 kodeketa izan behar duenez, karaktere kateak ere, defektuz, UTF-8 kodeketa erabiltzen dute. Karaktere katea gordetzen du.
- **`CharExprAST:`** UTF-8 kodeketarekin gordetako Unicode kode-puntu literal bat gordetzen du (gehienez 4 byte). Hortaz 32 biteko eta zeinurik gabeko zenbaki oso bat gordetzen du.

- **`UnaryExprAST:`** Eragigai bakarra duten eragiketa aritmetiko eta logikoak errepresentatzen ditu. Adibidez, `+5`, `-5`, eta `!true`. Eragigaia eta eragiketa mota gordetzen ditu.

- **`BinaryExprAST:`** Bi eragigai dituzten bitez bit-eko eragiketak eta eragiketa aritmetikoak errepresentatzen ditu. Horien artean batuketa, kenketa, biderketa, ezker- eta eskuin-desplazamenduak daude.

- **`BoolOpExprAST:`** '*and*' eta '*or*' eragiketa boolearrak errepresentatzen ditu. Ezker- eta eskuin-eragigaiak eta eragiketa mota gordetzen ditu.

- **`ComparisonExprAST:`** Konparaketa eragiketak errepresentatzen ditu. Horiek $<, \leq, >, \geq, =$ eta $\neq$ dira. Ezker- eta eskuin-eragigaiak eta eragiketa mota gordetzen ditu.

- **`CallExprAST:`** Funtzio-dei bat errepresentatzen du, funtzio-izena eta parametroak gordetzen ditu.

- **`VariableExprAST:`** Aldagai bat erabiltzen du. Exekutatzean aldagai horren balioa erabiliko da kalkulua egiteko.

- **`CastingExprAST:`** Balio bat mota batetik bestera bihurtzeko erabiltzen da. Helburu datu-mota eta bihurtu beharreko balioa gordetzen ditu.

- **`ArrayExprAST:`** Taula baten indize bat atzitzeko erabiltzen da. Taularen adierazpena eta erabilitako indizea gordetzen ditu.

- **`MemberAccessAST:`** Objektu baten eremu bat atzitzeko erabiltzen da. Objektuaren adierazpena eta eremuaren izena gordetzen ditu.

- **`AddressOfExprAST:`** Aldagai baten helbidea lortzeko balio du. Aldagaiaren adierazpena gordetzen du.

- **`DereferenceExprAST:`** Memoria-helbide batean dagoen balio bat irakurri eta idazteko balio du. Helbidea duen adierazpena gordetzen du.

- **`SizeofExprAST:`** Datu-mota baten tamaina bit-etan lortzeko balio du. Datu-mota gordetzen du.

- **`AsmExprAST:`** Mihiztadura lengoaia zatiak zuzenean Fleet-en sartu daitezke. Honekin behe-mailako kalkuluak eta kudeaketa egin daiteke. Adibidez, sistema eragilearekin zuzenean komunikatu. Honek **`AsmExprPart`** klasea erabiltzen du atal

desberdinak gordetzeko (sarrera parametroak, irteera parametroak, eta aldatutako erregistroak). 21 orrialdean ikusi daiteke honen erabileraren adibide bat.

3.2.4 Datu-motak

- **`Type:`** Datu-mota guztien oinarrizko klasea. Ez du informaziorik gordetzen, aldiz ondoren datozen bere azpiklaseek bakoitzak behar duen informazioa gordetzen du. Klase hau abstraktua da,
- **`PrimitiveType:`** Berezko motak errepresentatzen ditu. Datu mota gordetzen du.
- **`ArrayType:`** Taula bat errepresentatzen du. Elementu bakoitzaren mota, eta dimentsio bakoitzaren tamaina gordetzen ditu.
- **`NamedType:`** Izen batez identifikatutako mota bat errepresentatzen du. Hau soilik analisi sintaktikoaren fasean erabiltzen da. Izan ere, izenaz gain, ez du inolako informaziorik gordetzen, hortaz, ez da oso erabilgarria. Horregatik, datu-mota honetako balioak itzultzerakoan, beste mota batera ebatzi behar dira (oraingoz, izen batez soilik egiturak identifikatu daitezke).
- **`PointerToType:`** Datu-mota honetako balioek memoria-helbide bat gordetzen dute (erakusle bat, hain zuzen). Gainera, helbide horretan dagoen informazioaren mota ere gordetzen du datu-mota honek.
- **`StructDeclaration:`** Aurretik aipatutako klase honek, klase bat definitzeaz gain, aldagai baten mota ere errepresentatu dezake.

3.3 Sinbolo Taula

Programazio lengoaietan aldagaiak erabili ohi dira balioak gordetzeko. Hauek izen batez identifikatzen dira. Fleet-en, mota estatiko zorrotzak dituen lengoaia denez, aldagaiak erazagutu behar dira, konpiladoreak jakin dezan aldagai bakoitzak zein motatakoa den. Aldagai hori programan aurrerago aurkitzen denean, bere datu-mota jakiteko, *sinbolo taula* izeneko egitura bat erabili ohi da. Honetan, aldagai izen bakoitzari buruz daukagun informazioa gorde dezakegu. Baina ez aldagaiak bakarrik. Izan ere, funtzioak eta sortutako datu-motak ere beren izenaz identifikatu behar dira. Hortaz, atal honetan Fleet-en konpiladoreak erabiltzen duen sinbolo taula nola inplementatu den azalduko da.

Alde batetik, sinbolo taula bat ez da taula elkarkor baten oso desberdina. Aldagai bakoitzaren izenari, bere informazioa elkartu behar zaio. Hala ere, informazio desberdina gordeko denez (aldagaiak, funtzioak eta datu-motak), egitura bat sortzea egokia izan daiteke.

Egitura honi `SymbolTable` deitu diot. Honek 5 eremu nagusi ditu, taula elkarkorrak direnak. Hiru garrantzitsuenak `variables`, `functions` eta `types` dira, erazagututako aldagaiak, funtzioak eta motak gordetzen dituztenak, hurrenez hurren.

Horietaz gain, beste bi taula elkarkor erabiltzen dira. `ungenerated_functions` eta `ungenerated_types`. Bi hauek oraindik bitarteko kodera bihurtu ez diren funtzioak eta motak gordetzen dituzte, eta kode hila ezabatzeko erabiltzen dira. Izan ere, Fleet-en, inoiz deitzen ez diren funtzioei, eta inoiz erabiltzen ez diren datu-motei ez zaie inolako analisi semantikorik egiten, eta ez dira konpilatzen. Hau da, sintaktikoki zuzenak badira, ez dute arazorik emango. Gainera, honi esker konpilazioa azkarragoa da, ez baita kode hori bitarteko kodera itzuli behar, eta liburutegi estandar handia edukitzea ahalbidetzen du, exekutagarrien tamaina ahal den murritzena mantenduz.

Azkenik, programazio lengoaia egituratuetan bezala, Fleet-ek blokeen edo eremuen kontzeptua erabiltzen du, aldagaiak kapsulatzeko. Honela, bloke batean bi aldagaiek ezin dute izen bera izan, baina bloke desberdinetan ahalbidetzen da, aldagai desberdinak kontsideratzen direlako. Gainera, honi esker, aldagaiak ezin dira bere erazagupen-eremutik kanpo atzitu. Bloke hauek kabiatu daitezke, zuhaitz moduko egitura bat sortuz. Ondorioz, aldagai globalen eremua zuhaitzaren erroa izanda, aldagai baten erazagupena bilatzeko, uneko bloketik zuhaitzaren erroraino bilatu beharko litzateke. Hau inplementatzeko, sinbolo taula pila moduan egituratu daiteke. Non eremu berri batean sartzean, sinbolo taula berri bat sortzen den, eta sinbolo taulen pilan pilaratzen den. Honela, aldagai baten izena zein motatakoa den jakin nahi denean, pilaren gailurretik bilatzen hasten da, azken elementura iritsi arte.

Pila hori lista estekatu bakun bat bezala inplementatu da. Non `SymbolTable` klasearen instantzia bakoitzak, bere *gurasoaren* erreferentzia bat duen. Posible izango litzateke zuhaitz osoa eraikitzea, baina orokorrean hau ez da beharrezkoa konpilatu ahal izateko, zuhaitzean *anaiak* diren blokeak ez dutelako inoiz aldagairik konpartitzen.

Azkenik, aipatu behar da Fleet-en ezin direla ez funtzioak ez mota berriak funtzioen barnean sortu, hortaz, pila horretan erroa ez diren beste sinbolo taula guztiak ez dituztela ez funtzioen ez moten erazagupenik izango. Hala ere, pila osoa korritzen da, etorkizunean hauek ahalbidetzea interesgarria izan daitekelako.

3.4 LLVM IR-a sortzea

Behin Fleet kodea gure sintaxi zuhaitz abstraktu bihurtuta, LLVM-ren bitarteko kodea sortu behar da. Horretarako, 3.2.1 atalean deskribatutako klaseek `Generate` izeneko funtzio bat daukate. Funtzio hau klaseak errepresentatzen duena LLVM-ra bihurtu behar du. Hori lortzeko, bi aukera posible daude.

Alde batetik, LLVM IR-a testu moduan sortu daiteke. Hau honela eginez gero, posible da karaktere katearen erdian hasieratik ezagutzen ez den informazioa bete behar izatea. Honi *backpatching* deitzen zaio, eta oso ohikoa da honelako bihurketak egitean.

Beste aukera LLVM liburutegiak eskaintzen duen C++-en API-a erabiltzea da. Honek abantaila asko ditu. Alde batetik, ez dago karaktere kateen erdian ezer txertatzeko beharra, egitura memorian gordetzen delako, sintaxi zuhaitz baten antzera. Gainera, API-ak eskaintzen dituen funtzioek sententzia eta adierazpen bakoitzak hartu ditzaken parametro motak zorrozki definitzen ditu. Hortaz, akatsak sortzea zailagoa da.

Hori horrela, `Generate` funtzioak errekurtsiboki deitzen dira, sintaxi zuhaitzaren ostoetatik hasita, elementu guztiak LLVM IR-ra bihurtuz.

3.5 Erroreen tratamendua

Programa bat idaztean, akatsak egitea saihestezina da. Horregatik, konpiladore batean erroreak ondo tratatzea oso garrantzitsua da. Programa bat idaztean, lau errore mota nagusi sortu daitezke: errore lexikoak, sintaktikoak, semantikoak eta logikoak.

Errore lexikoak lengoaiak onartzen ez duen token bat ikustean gertatzen dira. Adibidez, identifikadore baten barnean ehuneko (‘%’) ikurra agertzen bada, eta lengoaiak letrak soilik onartzen dituenean.

Errore sintaktikoak programaren egituran errore bat dagoenean gertatzen dira. Adibidez, kode bloke bat ixten duen giltza (‘}’) ahaztuz gero, errore sintaktiko bat gertatu da, programak ez duelako eduki beharko lukeen egitura jarraitzen.

Errore semantikoak programaren esanahiari dagozkio. Zehazki, programa batean kontraesanak daudenean gertatu ohi dira. Adibidez, `int b = 0.2;` erazagupenak errore semantiko bat du. Izan ere, aldagaiaren datu mota osokoa da, baina esleitzen zaion balioa zenbaki arrazional bat da.

Errore logikoak konpiladore batek harrapatu ezin dituen erroreak dira. Programa ondo

idatzita dago, baina ez du garatzaileak nahi zuena egiten. Adibidez, aztertu dezagun ondorengo C kodea:

```
1 if (hero = 0)
2     printf("I need a hero");
```

Kode hori lexikoki, sintaktikoki eta semantikoki zuzena da. Programatzaileak `hero` aldagaiak zero balioa badu katea inprima dadin nahi du. Baina baldintzaren barruan bi berdin ikur (‘==’) erabili ordez, bakarra erabili du, adierazpena esleipen bihurtuz. C-n adierazpen baten balioa esleitzen ari den balioa denez, baldintza hori ‘`if(0)`’ jartzearen baliokidea da, eta, ondorioz, inoiz ez da mezua inprimatuko.

3.5.1 Analisi lexiko-sintaktikoa

Analisi lexiko eta sintaktikoa ANTLR-ek egiten du, eta horrek detektatzen ditu bertan gertatzen diren erroreak. Hortaz, errore hauek zuzenean inprimatzen dira, honen kokapenarekin batera.

Posible da ANTLR-en errore baten informazioa aztertzea, eta lengoaiaren ezaugarriak erabilita, errore-mezu hobeak sortzea. Hala ere, hau inplementatzeko denbora luzea beharko litzateke, eta lortutako onurak ez du konpentsatzen.

3.5.2 Analisi semantikoa

LLVM IR-a oso bitarteko kode zorrotza da. Edozein eragiketa egin ahal izateko, eragigai guztiak datu-mota berdinekoak izan behar dira. Hau ez bada betetzen, konpiladoreak ez du bere lana ondo egingo. Hortaz, Fleet konpiladoreak analisi semantiko sakona egin behar du, bereziki datu-motei dagokienez. Motekin zerikusia duten erroreez gain, beste batzuk ere gertatu daitezke. Adibidez, emaitza bat bueltatu behar duen funtzio batek emaitzarik ez bueltatzea, ezagutzen ez den funtzio edo aldagai bat atzitu nahi izatea eta identifikadore bat behin baino gehiagotan erazagutzea.

Errore guzti hauek errore semantikoak dira, eta sintaxi zuhaitza LLVM IR-ra bihurtzen diren bitartean detektatzen dira.

4. KAPITULUA

Fleet konpiladorearen erabilera

Fleet-en konpiladorearen bertsio desberdinak kudeatzeko Git erabili da. Zerbitzari moduan, GitLab aukeratu da, eskaintzen dituen aukerengatik. Izan ere, GitLab-en *Continuous Integration* (CI) izeneko metodologia aplikatu daiteke.

Metodologia honen arabera, garatzaileek idatzitako kodea ia egunero bateratu behar da, dena funtzionatzen duela ziurtatzeko. Oraingoz Fleet-ek garatzaile bakarra du, hortaz horrek ez du zailtasun handirik, baina CI metodologiak beste praktika batzuk ere gomendatzen ditu. Adibidez, egiten den *commit* bakoitza konpilatu eta probatu behar dela.

Horretarako, GitLab konfiguratu daiteke commit bat igotzen den bakoitzean, kodea konpilatzeko. Horretarako, '`.gitlab-ci.yml`' fitxategian kodea nola konpilatu azaldu behar da. Fleet-en konpiladoreak CMake erabiltzen duenez proiektua kudeatzeko eta konpilatzeko, eta horren pareko den CTest probak automatikoki egiteko, fitxategi hori sinplea eta laburra da.

Gerta daitezken arazoak dependentziak dira. Izan ere, Fleet-en konpiladoreak kanpodependentziak ditu. Adibidez, LLVM-ren liburutegi eta exekutagarriak instalatuta eduki behar ditu. Hortaz, CI prozesua sinplifikatzeko, Docker irudi bat erabili da. Docker makina birtualen antzekoak diren baina askoz arinagoak diren *edukiontziak* sortzen ditu. Honela, kodea konpilatzeko beharrezkoak diren fitxategiak zuzenean edukiontzi batean gordetzen dira, eta GitLab-ek edukiontzi hori hartu, eta bertan konpilatu dezake kodea.

Prozesu honi, eta proba programak idatzi izanari esker, proba horiek emaitza egokia lortzen dutela bermatu daiteke, commit bat egiten den bakoitzean.

Ondorengo ataletan, bakoitzak bere ordenagailuan Fleet konpiladorea instalatzeko jarraitu

behar dituen urratsak daude. Kontuan izan konpiladorea soilik Linux-etik soilik konpilatu daitekela, eta bakarrik x86_64 motako CPU batean probatu dela.

4.1 Dependentziak instalatzea

Konpiladore bat sortzea lan konplexua da. Horregatik, lana errazteko, Fleet konpiladoreak hainbat liburutegi erabiltzen ditu, eta kodea konpilatu ahal izateko, liburutegi horiek instalatu behar dira.

Ondorengo agindua Debian-en oinarritutako (Ubuntu barne) linux banaketetan behar diren dependentziak instalatuko ditu (Java ANTLR erabili ahal izateko, LLVM back-end-a, eta Python 3, probak automatikoki exekutatzeko).

```
curl -L https://goo.gl/Zw1aC4 | sudo bash
```

4.2 Kodea lortu

Fleet konpiladorearen kodea `https://gitlab.com/aritzh/FleetLang` biltegian dago. Hau probatu ahal izateko, deskargatu behar da, eta horretarako bi era nagusi daude. Lehenengoa web bitartez egiten da. Goiko estekan deskargatzeko botoiari emanez, kodea konprimatuta deskargatuko da.

Bigarrena, `git` erabilita, terminaletik egiten da. Horretarako, lehenengo nahi dugun karpetara joan, eta ondorengo agindua exekutatuz gero, kodea deskargatuko da.

```
git clone --depth 1 https://gitlab.com/aritzh/FleetLang.git
```

Honek *FleetLang* izeneko karpeta bat sortuko du, eta hor barruan egongo da kode guztia.

4.3 Kodea konpilatu

Kodea konpilatzea oso erraza da, proiektuak CMake erabiltzen duelako. Ondorengo aginduek irteerako karpeta bat sortzen dute, eta kodea konpilatzen dute.

```
cd FleetLang
mkdir build && cd build
cmake ..
make -j4
```

Arazorik ez bada sortzen, horrek konpiladorea konpilatu beharko luke. Agindu horiek amaitzean, sortutako `build` karpetan `FleetLang` izeneko exekutagarria egongo da. Hori da Fleet lengoaiaren konpiladorea.

4.4 Konpiladorearen erabilera

Atal honetan konpiladorea nola erabiltzen den azalduko da. Honetarako, konpiladoreak har ditzakeen parametroak aztertuko dira.

Aukerak

- **`-v` edo `--verbose`:** Konpiladoreak irteera gehiago sortzen du. Honen gabe, erroreak bakarrik erakusten dira; honekin, aldiz, informazio gehiago pantailaratzen da. Adibidez, definitutako funtzioak, analizatzen diren fitxategiak, etab.
- **`-r` edo `--release`:** Parametro hau pasa gabe, LLVM-k ez du kodea optimizatuko. Parametroa pasaz gero, LLVM-k hainbat optimizazio gauzatzen ditu. Adibidez, konstanteen hedapena (adibidez, 1+2 adierazpena, 3 balioarekin zuzenean ordezkatzea), kode hila ezabatzea, eta memoria erabiliz egiten diren kalkulu batzuk zuzenean erregistroak erabilita egitera aldatzea.
- **`-s` edo `--single`:** Parametro hau pasa gabe, konpiladoreak direktorio oso batean dauden fitxategiak konpilatzen ditu. Parametroa pasaz gero, fitxategi bakar bat konpilatzen du.
- **`-i` edo `--print-llvm`:** Sarrerako kodea konpilatzean sortu den LLVM IR-a irteera estandarrean inprimatu.
- **`-c` edo `--only-compile`:** Kodea ez estekatzeko. Erabilgarria kode osoa konpilazio bakarrean egin ezin denean. Adibidez beste liburutegi batekin edo C kodearekin estekatzeko.

Parametroak

- **`-l KARPETA` edo `--stdlib KARPETA`:** Fleet liburutegi estandarraren kokapena. Konpiladorearen kodearen `src/fleet` karpeta izan behar da, liburutegi estandarra erabili nahi bada (oraingoz liburutegi estandarrak oinarrizko ezaugarriak besterik ez ditu, baina balioak pantailaratzeko, eta memoria alokatzeko balio du).

- **`-o FITXATEGIA` edo `--output FITXATEGIA` (beharrezkoa):** Konpilazioaren emaitza gordeko duena fitxategiaren izena. Besterik adierazi ezean, exekutagarria gordeko du. `-c` aukera pasaz gero, estekatu gabeko objektu fitxategia gordeko du.

- **`-t PLATAFORMA` edo `--triple PLATAFORMA`:** LLVM-k beste plataforma batera-ko konpila dezan pasa daiteken parametroa. Formatua LLVM-rena propioa da, eta ez dago formatu bakarra (helburu CPU mota bakoitzak formatu desberdina erabili dezake). Hala ere, ondorengo biak probatu dira:

  - 64 biteko ordenagailu arrunt bat, linux-ekin: `x86_64-pc-linux-gnu`
  - Raspberry Pi 3 bat, Raspbian-ekin: `armv6-unknown-linux-gnueabihf`

Parametro horietaz gain, konpilatu nahi den karpeta edo fitxategia ere pasa behar zaio, baina horrek ez du aurretik parametro berezirik behar.

Ondoren erabileraren adibide batzuk azalduko dira.

- `src` karpetan dauden fitxategi guztiak `irteera` izeneko exekutagarri batean konpilatzeko, ondorengo agindua nahikoa da:

  ```
  ./FleetLang src -o irteera -l ../src/fleet
  ```

- Fitxategi bakar bat konpilatzeko:

  ```
  ./FleetLang input.fl -o output -l ../src/fleet
  ```

- Fleet kodea C-rekin batera estekatzeko bi aukera daude. Izan ere, Fleet-en liburutegi estandarra ez da C-renarekin bateragarria. Horregatik, bi liburutegi estandarretatik bat aukeratu behar da. Ondorengo adibideetan suposatuko da GCC erabiltzen dela C konpiladore moduan.

  - C-ren liburutegi estandarra erabiliz:

    ```
    gcc -c input.c -o cobject.o
    ./FleetLang -cs input.fl -o flobject.o
    gcc flobject.o cobjekt.o -o output
    ```

  - Fleet-en liburutegi estandarra erabiliz:

```
gcc -c -nostdlib input.c -o cobject.o
./FleetLang -cs input.fl -o flobject.o -l ../src/fleet
gcc -nostdlib flobject.o cobjekt.o -o output
```

Kontuan izan orokorrean programa berdinak ezin direla bi eratara konpilatu. Izan ere, batek C-ren liburutegiak eskaintzen dituen `printf` moduko funtzioak erabili ahal izango ditu, eta besteak, aldiz Fleet-en liburutegiak eskainitako `println` modukoak erabili beharko ditu.

Kontuan edukitzeko beste atal bat optimizazioarekin zerikusia du. Aurretik aipatu den bezala, Fleet-en deitzen ez diren funtzioak ez dira konpilatzen. Honek arazoa eman ditzake funtzio horiek C fitxategi batetik deitzen badira (Fleet konpiladoreak ezin duelako hori jakin). Hortaz, Fleet konpiladoreak bere liburutegi estandarra erabiltzen ez duenean, optimizazio hau baztertzen du.

Azkenik, 3. kapituluan aipatu den bezala, gogoratu behar da Fleet-en funtzioen izenak aldatzen direla, bere parametroen arabera. Hortaz, C-tik deitu ahal izateko, `@foreign` oharra gehitu, edo funtzioaren izena '#' ikurraz hasi behar da. Honela funtzioaren izena ez da aldatuko, eta C-tik deitu ahal izango da.

5. KAPITULUA

Ondorioak eta etorkizunerako lana

Azken atal honetan proiektuaren ondorio nagusiak, eta etorkizunean proiektua garatzen jarraitzeko jarraitu litezkeen ideiak deskribatzen dira.

5.1 Ondorioak

Konputazioa adarreko irakasgaiei esker (bereziki Konpilazioa irakasgaia), proiektu honen helburu nagusiak bete dira. Izan ere, Fleet problema batzuk ebazteko erabili daiteken programazio lengoaia bat da.

Hala ere, programazio lengoaia bat sortzea lan zabal eta zaila da, dokumentazioa murritza delako, eta Internet-en sakabanatuta dagoelako. Orokorrean, ez dira programazio lengoaia asko sortzen, hauen garapena merkatu-nitxo bat bihurtuz. Hortaz, oso gutxik dokumentatzen dute garapenean izandako arazoak nola konpondu dituzten. Adibidez, zaila gerta daiteke aurkitzea LLVM IR-n taulako elementu bat nola atzitzen den, edo konpilatutako fitxategia exekutagarri batean estekatzeko prozedura, nahiz eta LLVM-ren dokumentazioa oso zabala den. Horretarako, garapenean dauden eta kode irekikoak diren hainbat proiektuetatik hartu da inspirazioa. Hau honela, proiektu honek aurretik kontsideratu ez den beste helburu bat bete dezake: LLVM-ren front-end baten adibide izatea, etorkizunean sortu daitezkeen beste programazio lengoaietarako.

Nolanahi ere, proiektuaren irismenean definitutako ezaugarri minimoak, eta gehigarri batzuk ere betetzea lortu da. Gainera, bai ohiko x86_64 CPU batean, bai ARM motako Raspberry Pi baterako kodea konpilatu daitekela ikusi da, C-ren liburutegi estandarra erabili gabe.

Pertsonalki, proiektu honek ezagutza berri asko eskaini dizkit. Alde batetik, programazio lengoaia konpilatu bat nola sortzen den irakatsi dit, eta bidean C++-en, LLVM-ren, ANTLR-en eta erabili ditudan beste tresna askoren ezagutzak zabaldu dizkit.

5.2 Etorkizunerako lana

Programazio lengoaia bati beti sartu dakizkioke ezaugarri berriak, baita eraginkortasuna hobetu ere. Hortaz, atal hau Fleet konpiladorean garatzear dauden ezaugarrietan, eta denbora gehiago izanez gero oinarrizkoak izan zitezkeen ezaugarrietan zentratuko da.

Alde batetik, interesgarria izango litzateke Fleet kodeak sortutako exekutagarri bat araztu ahal izatea, GDB erabilita, adibidez. Horretarako LLVM-k API bat eskaintzen du, eta horretarako lana hasita dago dagoeneko, baina ez da erabilgarria oraindik.

Beste aldetik, errore batzuk hobeto harrapatzea eta erabiltzaileari erakustea interesgarria izango litzateke. Izan ere, LLVM-ren bitarteko kodea oso zorrotza da (datu motekin bereziki).

Gainera, Fleet-en liburutegi estandarra zabaldu daiteke. Oraingoz ezinbestekoak kontsideratu diren ezaugarriak ditu: zenbakiak eta karaktere kateak inprimatzea. Horiek inplementatzeko kateen luzera, 10 oinarriko logaritmoa eta berreketa ere inplementatu dira.

Interesgarria izan daiteken beste ezaugarri bat IDE-etan integratzea da. Oraingoz testu fitxategi bat bezala editatu da Fleet kodea, inolako kolore edo auto-betetzerik gabe, orain arte idatzitako kodea ez delako tamaina handiko proiekturik izan, eta orokorrean nahiko sinplea izan delako.

Fleet-ek egiturak sortzeko ahalmena badu ere, bjektuei orientatutako programazio paradigma ahalbidetu daiteke, kodeak egitura eta kapsulaketa hobea lortu dezan.

Azkenik, pare bat funtzionalitate erdi-inplementatuta daude. Honen arrazoi nagusia denbora falta izan da. Hauen adibide taulen atzipena, eta programak arazteko ikurrak sortzea daude. Bi ezaugarri hauek gratazen hasi diren arren, oraindik ez daude ondo funtzionatzeko moduan. Horregatik hauek amaitzea beste ezaugarri berri batzuk guztiz inplementatzea baino errazagoa izan daiteke.

Eranskinak

A. ERANSKINA

Gramatika

```
grammar FleetLang;  // Gramatikaren izena

program  // Lehenengo erregela hasierako ikurra da
    : topLevelDecl* EOF
    ;

topLevelDecl               // Programaren erroan agertu daitezkeen egiturak
    : function             // Funtzioen erazagupen edota definizioak
    | typeDecl             // Egituren definizioak
    | declaration ';'  // Aldagai globalen erazagupen edota hasieraketa
    ;

typeDecl  // Egitura baten definizioa
    : annotation* CLASS ID '{' (declaration ';')* '}'
    ;

function  // Funtzio baten erazagupen edota definizioa
    : annotation* PUBLIC? (type | voidType) (ID|FOREIGN_FUNC_ID) '(' (type ID ('
        ↪ ,' type ID)* )? ')'
      ( ('{' statement* '}') | ';')
    ;

annotation  // Oharrak. Egitura eta funtzioetan meta-datuak gordetzeko
    : '@' ID ('(' STRING_LIT (',' STRING_LIT)* ')')?
    ;

type  // Datu motak
    : ID                       // Sinbolo taulan gordetakoak, oraingoz egiturak.
    | primitiveType            // Oinarrizko datu motak
    | '*' type                 // Beste mota bati erakuslea
    | type ('[' INT_LIT ']')+  // Taulak
    ;

primitiveType
    : INT              // Zenbaki osokoak
    | UINT             // Zenbaki arruntak
    | FLOAT            // Zehaztasun bakuneko koma higikorreko arrazionalak
    | DOUBLE           // Zehaztasun bikoitzeko koma higikorreko arrazionalak
    | BOOLEAN          // Balio boolearra
    | SIZED_INT_TYPE   // Tamaina finkoko zenbaki oso eta arruntak
    ;

voidType     // Balio ezaren mota
```

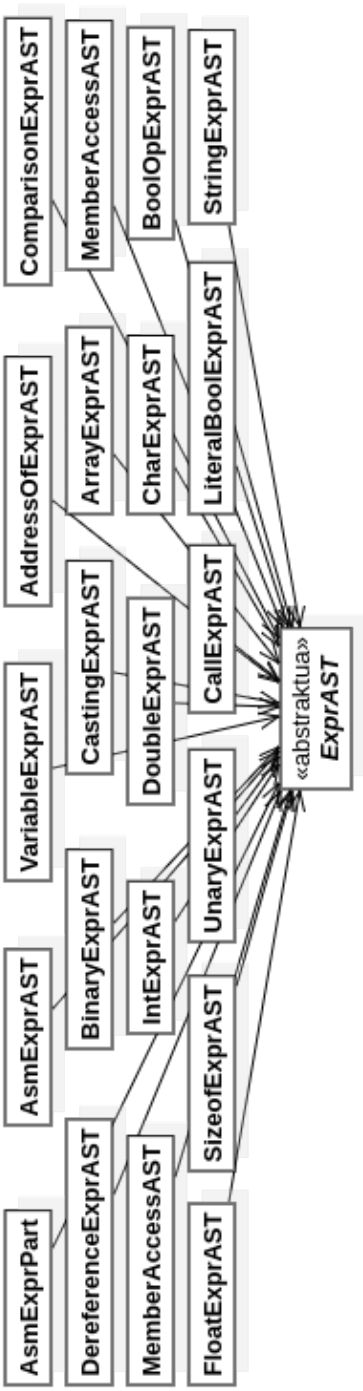
```
43      : VOID
44      ;
45
46  statement
47      : IF '(' expression')' statement (ELSE statement)? // Baldintza
48      | WHILE '(' expression ')' statement  // While begizta
49      | RETURN expression? ';'  // Return sententzia
50      | '{' statement* '}'  // Kode blokeak
51      | STATIC? declaration ';'  // Aldagaien erazagupenak
52      | assignmentStmt ';'  // Esleipenak
53      | expression ';'  // Sententzia moduan erabilitako adierazpena
54      | FOR '(' declaration? ';' expression? ';' assignmentStmt? ')' statement  //
          ↪  For begizta
55      | DO statement WHILE '(' expression ')' ';'  // Do-while begizta
56      | ';'  // Sententzia hutsa, while hutsak sortzeko, adibidez.
57      ;
58
59  assignmentStmt  // Esleipenak
60      : expression assignmentOperator expression // Ohiko esleipenak
61      | expression (INCREMENT|DECREMENT)  // Balioa inkrementatu eta
            ↪ dekrementatzeko
62      ;
63
64  assignmentOperator  // Esleipena eta eragiketa modu laburrean egiteko
65      : ('+='|'-='|'*='|'/='|'&&='|'||='|'&='|'|='|'^='|'%='|'=');
66
67  declaration  // Aldagaien erazagupenak (hautazko hasieraketarekin)
68      : type ID ('=' expression)?
69      ;
70
71  expression  // Adierazpenak
72      : INT_LIT     // Osoko literalak
73      | DOUBLE_LIT  // Zehaztasun bikoitzeko koma higikorreko literala
74      | FLOAT_LIT   // Zehaztasun sinpleko koma higikorreko literala
75      | STRING_LIT  // Karaktere kate literala
76      | CHAR_LIT    // Karaktere literala
77      | (TRUE | FALSE)  // Boolear literala
78      | '(' type ')' expression  // Mota aldaketa
79      | ID  // Erazagututako aldagai bat
80      | expression '.' ID  // Egitura baten eremuari atzipena
81      | expression '[' expression ']'  // Taulen atzipena
82      | ('+'|'-') expression  // Eragigai bakarreko eragiketa aritmetikoak
83      | '!' expression  // Eragigai bakarreko eragiketa logikoak
84      | SIZEOF '(' type ')'  // Datu mota baten tamaina lortzeko adierazpena
85      | ID '(' (expression (',' expression)*)? ')'  // Funtzio deiak
86      | '&' expression  // Aldagaien helbidea lortzeko
87      | '*' expression  // Helbide batean dagoen balioa lortzeko
88      | expression ('*'|'/'|'%') expression  // 1. mailako eragiketa bitarrak
89      | expression ('+'|'-') expression  // 2. mailako eragiketa bitarrak
90      | expression ('>>>'|'>>'|'<<') expression  // 3. mailako eragiketa bitarrak
91      | expression ('<='|'>='|'>'|'<') expression  // 4. mailako eragiketa
            ↪ bitarrak
92      | expression ('=='|'!=') expression  // 5. mailako eragiketa bitarrak
93      | expression '&' expression  // 6. mailako eragiketa bitarrak
94      | expression '^' expression  // 7. mailako eragiketa bitarrak
95      | expression '|' expression  // 8. mailako eragiketa bitarrak
96      | expression '&&' expression  // 9. mailako eragiketa bitarrak
97      | expression '||' expression  // 10. mailako eragiketa bitarrak
98      | '(' expression ')'  // Lehentasuna aldatzeko parentesiak
99      | ASM '(' STRING_LIT (    // Mihiztadura lengoaiaren zatiak sartzeko
100       ':' (asmExprPart (',' asmExprPart)* )? (
101       ':' (asmExprPart (',' asmExprPart)* )? (
102       ':' STRING_LIT? (',' STRING_LIT)* )? )? )? ')'
103     ;
104
105 asmExprPart
```

```
106     : STRING_LIT '(' expression ')'
107     ;
108
109 CLASS: 'class';
110
111 UINT: 'uint';
112 INT: 'int';
113 FLOAT: 'float';
114 DOUBLE: 'double';
115 BOOLEAN: 'bool';
116 VOID: 'void';
117
118 IF: 'if';
119 WHILE: 'while';
120 RETURN: 'return';
121 ELSE: 'else';
122 FOR: 'for';
123 DO: 'do';
124
125 TRUE: 'true';
126 FALSE: 'false';
127
128 PUBLIC: 'pub';
129 STATIC: 'static';
130
131 SIZEOF: 'sizeof';
132
133 ADDRESS: '&'; // And bitwise and
134
135 INCREMENT: '++';
136 DECREMENT: '--';
137
138 LEFT_SHIFT: '<<';
139 LOGICAL_RIGHT_SHIFT: '>>>';
140 ARITHMETIC_RIGHT_SHIFT: '>>';
141
142 LESS_EQUAL: '<=';
143 GREATER_EQUAL: '>=';
144 LESS_THAN: '<';
145 GREATER_THAN: '>';
146 EQUALS: '==';
147 NOT_EQUALS: '!=';
148
149 ADD_ASSIGN: '+=';
150 SUB_ASSIGN: '-=';
151 MUL_ASSIGN: '*=';
152 DIV_ASSIGN: '/=';
153 MODULO_ASSIGN: '%=';
154 BOOL_AND_ASSIGN: '&&=';
155 BOOL_OR_ASSIGN: '||=';
156 BIT_AND_ASSIGN: '&=';
157 BIT_OR_ASSIGN: '|=';
158 BIT_XOR_ASSIGN: '^=';
159
160 BOOL_OR: '||';
161 BOOL_AND: '&&';
162
163 ADD: '+';
164 SUBTRACT: '-';
165 MULTIPLY: '*';
166 DIVIDE: '/';
167 BIT_OR: '|';
168 BIT_XOR: '^';
169 MODULO: '%';
170
171 ASSIGN: '=';
172
```

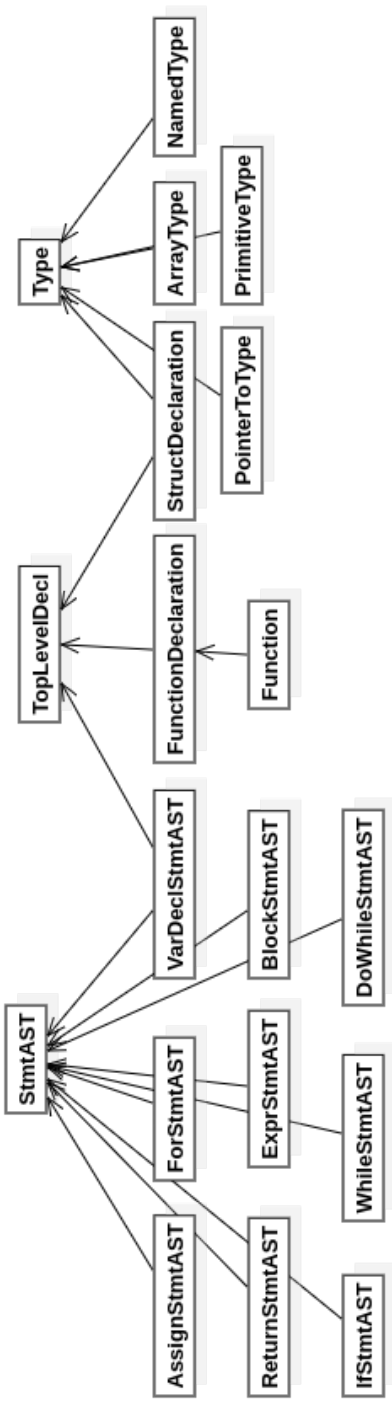
```
173
174 LPAREN: '(';
175 RPAREN: ')';
176
177 AT: '@';
178
179 ASM: 'asm';
180
181 SIZED_INT_TYPE: [iu]('8'|'16'|'32'|'64'|'128');
182 ID:           [_]?[a-zA-Z][a-zA-Z_0-9]*;
183 FOREIGN_FUNC_ID: '#' ID;
184
185 FLOAT_LIT:  [0-9]*'.'[0-9]+([eE][+\-]?[0-9]+)?'f';
186 DOUBLE_LIT: [0-9]*'.'[0-9]+([eE][+\-]?[0-9]+)?;
187 INT_LIT:    ('0'[oObBxX]?)?[0-9]+([iu]('8'|'16'|'32'|'64'|'128'))?;
188 STRING_LIT: '"' StringCharacter* '"';
189 CHAR_LIT:   '\'' StringCharacter '\'';
190
191 DOT: '.';
192
193 // Baztertu beharreko tokenak, hauek ez
194 // dira analisi sintaktikora pasatzen
195 WS:           [ \t\r\n]+ -> skip;
196 COMMENT:      '/*' .*? '*/' -> skip;
197 LINE_COMMENT: '//' ~[\r\n]* -> skip;
198
199 // Karaktere literala.
200 // Adib.: a, \t, \n, \\, \'
201 fragment Character
202     : ~['\\]
203     | '\\' [btnfr"'\\]
204     ;
205
206 // Karaktere-kateen karaktere literala.
207 // Adib.: a, \t, \n, \\, \"
208 fragment StringCharacter
209     : ~["\\]
210     | '\\' [btnfr"'\\] // Escape sequence
211     ;
```

B. ERANSKINA

Klase-diagramak



B.1 Irudia: Adierazpenen klase diagrama.



B.2 Irudia: Sententziak, motak eta goi-mailako egituren klase-diagrama.

Erreferentziak

1. *IEEE Standard for Binary Floating-Point Arithmetic* Standard (Institute of Electrical and Electronics Engineers, Dec. 1985). doi:10.1109/IEEESTD.1985.82928 (28 orrialdean).
2. Crocker, D. & Overell, P. *Augmented BNF for Syntax Specifications: ABNF* STD 68. http://www.rfc-editor.org/rfc/rfc5234.txt (RFC Editor, Jan. 2008). http://www.rfc-editor.org/rfc/rfc5234.txt (24 orrialdean).
3. *Information technology – Syntactic metalanguage – Extended BNF* Standard (International Organization for Standardization, Dec. 1996) (24 orrialdean).
4. Jinks, P. *Notations for context-free grammars* http://www.cs.man.ac.uk/~pjj/bnf/bnf.html#EBNF (24 orrialdean).
5. Parr, T. *Adaptive LL(*) Parsing: The Power of Dynamic Analysis* http://www.antlr.org/papers/allstar-techreport.pdf (4 orrialdean).

Bibliografia

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman *Compilers: Principles, Techniques, and Tools.* 1986.

[2] LLVM Developers *LLVM Language Reference Manual* 2018 https://llvm.org/docs/LangRef.html