Зборник радова Факултета техничких наука Нови Сад

UDK: 004.42:004.738.12
DOI: https://doi.org/10.24867/33BE29Bjelica

ИЗРАДА КОМПАЈЛЕРА УПОТРЕБОМ РАСТЕМО БИБЛИОТЕКЕ

COMPILER DEVELOPMENT USING THE RUSTEMO LIBRARY

Марко Бјелица, *Факултет техничких наука, Нови Сад*

Област – РАЧУНАРСТВО И АУТОМАТИКА

Кратак садржај – *У овом раду је уз ослонац на библиотеку Rustemo, изграђен компајлер Расти, назван тако јер је имплементиран у програмском језику Rust. У раду је дата теоријска основа за све фазе компајлирања које су реализоване у Растију, а то су лексичка, синтаксна и семантичка анализа, са највећим фокусом на синтаксну анализу. Расти се састоји од лексичког анализатора (имплементираног ручно и изгенерисаног помоћу Rustemo библиотеке), синтаксног анализатора (имплементираног ручно и изгенерисаног Rustemo библитеком), семантичког анализатора и евалуатора.*

Кључне речи: *Компајлер, Rustemo, Rust, лексички анализатор, синтаксни анализатор, семантички анализатор, евалуатор.*

Abstract – The paper presents a compiler, named Rasti, developed witth the *Rustemo library. Compiler name Rasti was chosen, because it is implemented in the Rust programming language. This work provides the theoretical foundation for all phases of compilation realized in Rasti, namely lexical, syntax and semantic analysis, with the greatest focus on syntax analysis. Rasti consists of a lexical analyzer (implemented manually and generated with Rustemo), a syntax analyzer (also implemented manually and generated with Rustemo), a semantic analyzer and an evaluator.*

Keywords: *Compiler, Rustemo, Rust, lexical analyzer, syntax analyzer, semantic analyzer, evaluator.*

1. УВОД

У свијету рачунарства, компајлер је програм који преводи изворни код написан на високом нивоу у циљни код погодан за извршавање на рачунару или виртуелној машини. Изворни код може бити написан у програмским језицима попут *C*, *C++*, *Rust*, *Java*, *Python* или чак у асемблеру. Циљни код може бити неки други програмски језик (најчешће нижег нивоа у односу на изворни), машински код, асемблер, бајткод или нека међуформа.

Процес компајлирања се састоји од неколико узастопних фаза, при чему свака фаза има одређен задатак, а то су:

НАПОМЕНА:
Овај рад проистекао је из мастер рада чији ментор је био др Игор Дејановић, ред. проф.

лексичка анализа, синтаксна анализа, семантичка анализа, генерисање међукода, оптимизација и на крају генерисање циљног кода.

У оквиру овог рада представљен је Расти, компајлер изграђен у програмском језику *Rust*, уз помоћ библиотеке *Rustemo* [1]. Фокус је на теоријској основи компајлирања и практичној имплементацији лексичке, синтаксне и семантичке анализе. У пројекту лексичка анализа је реализована на два начина, а то су ручном имплементацијом и генерисањем помоћу *Rustemo* библиотеке. Слично томе синтаксна анализа је реализована ручном имплементацијом парсера са рекурзивним спустом и аутоматски изгенерисаним LR парсером на основу прослијеђене граматике *Rustemo* библиотеци. Затим семантичка анализа обрађује синтаксно стабло добијено од парсера, провјерава логичку исправност и гради ново стабло. Након тога, евалуатор пролази кроз стабло настало семантичком анализом и извршава наредбе.

Расти подржава рад са изразима који садрже нумеричке и булове вриједности, као и комбиновање истих са аритметичким, логичким, релационим и унарним операторима. Такође, подржане су следеће наредбе: декларација промјењивих, додјела вриједности, петље и условне наредбе.

Рад је структуриран у четири дијела. Први дио представља наведени увод у којем је укратко описан концепт компајлера и које фазе има, такође дат је кратак осврт на Расти. У другом дијелу је теоријски преглед основа компајлера и опис фаза лексичке, синтаксне и семантичке анализе. Затим у трећем дијелу је описан Расти, кроз његове функционалности, архитектуру и разлике које изгенерисани и ручно имплементирани парсер носе. И на крају у четвртом дијелу дат је закључак у којем су разматране могуће примјене и будући правци развоја.

2. ТЕОРИЈСКЕ ОСНОВЕ

2.1. Процес компајлирања

За успјешно рјешавање проблема из пословног домена, неопходно је прецизно дефинисање проблема уз коришћење одговарајућег језика. Док за комуникацију међу људима користи се природни језик, рачунари нису у могућности да га интерпретирају на начин, на који то људи раде. Због тога је потребно користити језик који рачунари могу да разумију и формализовати проблем у складу са правилима тог језика.

Због сложености машинског језика за људе, развијени су програмски језици који омогућавају једноставнију комуникацију са рачунарима. Изворни код написан у неком програмском језику се не може директно извршавати на рачунару, јер рачунар разумије само инструкције у машинском коду. Због тога потребно је превести изворни код у облик који рачунар може разумијети и извршити. Тај превод обавља компајлер, који као улаз прихвата изворни програм, а као излаз генерише циљни код, и тај процес се зове компајлирање.

Сваком програмском језику потребан је преводилац, али неки језици нису погодни за све задатке, па се у неким доменима јавља потреба за креирањем нових, прилагођених језика. То су језици специфични за домен (енг. *Domain-specific languages*), осмишљени за рјешавање проблема у конкретном домену и за чију изградњу је потребно разумијевање процеса компајлирања.

2.2. Компајлери и интерпретери

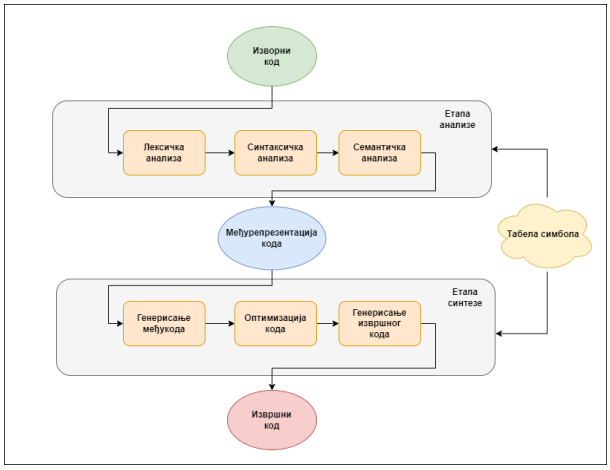
Први корак у рјешавању проблема из пословног домена, формулисаног терминологијом програмског језика, јесте превођење изворног у извршни код. Начин превођења може да се разликује у зависности да ли се користе компајлери или интерпретери [2].

Компајлери у потпуности преводе изворни код у извршни код, који је затим спреман за покретање. Овај приступ омогућава оптимизацију кода прије извршавања, што резултује бржим радом програма.

Интерпретери обрађују код инструкцију по инструкцију, без стварања извршне датотеке. Спорији приступ, али омогућава бољу дијагностику проблема, управо зато што се програм прати током извршавања.

2.3. Етапе компајлирања

Компајлирање је сложен процес који се састоји из више узастопних фаза. Те фазе се дијеле на двије главне етапе, а то су анализа и синтеза, као што је приказано на слици 1.



Слика 1. *Графички приказ компајлирања*

Етапа анализе или предњи дио (енг. *frontend*) компајлера обрађује изворни код и претвара га у међурепрезентацију, као што је апстрактно синтаксно стабло. Ова етапа укључује три фазе: лексичка, синтаксна и семантичка анализа, које провјеравају исправност кода према правилима језика.

Етапа анализе или задњи дио (енг. *backend*) компајлера користи међурепрезентацију за генерисање извршног кода.

Ове етапе нису строго раздвојене, већ међусобно сарађују путем табеле симбола, која чува податке о идентификаторима и игра кључну улогу у компајлирању.

2.4. Лексичка анализа

Лексичка анализа је прва фаза компајлирања, чији је задатак да чита изворни код карактер по карактер и претвара га у низ токена. Овај процес обавља лексички анализатор или лексер који:

- Препознаје лексеме и класификује их по типу градећи токене од њих.
- Прескаче непотребне елементе попут коментара и празног простора (ово је опционо, јер постоје језици гдје увлачења имају значење).
- Памти позицију токена у коду, ради лакшег лоцирања грешке уколико се деси.
- Гради табелу симбола са подацима о индентификаторима.
- Пријављује грешку, уколико не може да препозна улазни симбол.

Лексичка анализа је једина фаза која ради директно са карактерима, све наредне фазе раде са токенима.

2.5. Синтаксна анализа

Синтаксна анализа је друга фаза компајлирања која слиједи након лексичке анализе. Слично као што синтакса природног језика проучава правила која одређују како се ријечи комбинују у реченице, тако граматика програмског језика представља скуп правила који дефинише, како се токени могу комбиновати да би се добио исправно структуриран код.

Током ове фазе, синтаксни анализатор или парсер на основу токена, добијених из лексичке анализе, провјерава да ли њихов редослијед у складу са граматиком језика, притом градећи синтаксно стабло [3].

Синтаксно стабло или стабло парсирања (енг. *parse tree*) јесте хијерархијска структура која одражава организацију кода у складу са граматиком програмског језика. Формира се током синтаксне анализе и служи као основа за наредну фазу компајлирања, што је семантичка анализа.

Елементи синтаксног стабла:

- Чворови стабла могу бити терминали или нетерминали.
- Коријен стабла је почетни симбол граматике.
- Листови стабла су увијек терминали, јер се не могу даље расчланити.

У зависности од начина изградње синтаксног стабла, односно правца у којем парсер се креће приликом препознавања структуре програма, постоје два типа синтаксне анализе, а то су синтаксна анализа наниже (енг. *top-down parsing*) и синтаксна анализа навише (енг. *bottom-up parsing*).

2.5.1. Синтаксна анализа наниже

Синтаксна анализа наниже је приступ парсирању који започиње од почетног нетерминала граматике и покушава да конструише синтаксно стабло према доњим листовима, односно терминалима. Анализатор покушава да усклади улазне токене са продукцијама граматике, примјењујући правила тако да генерише улазни низ. У сваком кораку, одлука о примјени правила доноси се на основу предувидног симбола (енг. *lookahead*) [4].

Најзаступљенији парсери који користе ову методу су:

- Парсер са рекурзивним спустом (ручно имплементиран парсер у Растију).
- LL(1) парсер.

Овај приступ није отпоран на појаву лијеве рекурзије у граматици.

2.5.1.1. Рекурзивни спуст

Техника рекурзивног спуста представља приступ синтаксној анализи наниже, при којем се сваки нетерминални симбол граматике реализује као засебна рекурзивна функција. На основу предувидног симбола, ове функције селективно примјењују одговарајућа продукцијска правила. Терминални симболи се директно пореде са тренутним улазом, а у случају непоклапања јавља се синтаксна грешка.

2.5.2. Синтаксна анализа навише

Анализа навише представља процес конструисања синтаксног стабла за дати улазни низ токена, полазећи од његових листова и поступно сводећи га ка коријену, односно ка почетном симболу граматике. Током овог процеса, у сваком кораку анализе, неки подниз улаза који одговара десној страни правила граматике замјењује се његовом лијевом страном. Ако су ови поднизови изабрани на правилан начин, процес парсирања одговара обрнутом току крајњег десног извођења, гдје се продукције примењују од листова ка коријену стабла

2.5.2.1. Анализа навише пребацивањем и свођењем

Анализа навише пребацивањем и свођењем (енг. *shift-reduce parsing*) је општи облик синтаксне анализе навише која гради синтаксно стабло од листова ка коријену []. Овај приступ користи стек за праћење парцијалних резултата обраде и има двије методе:

- Пребацивање (енг. *shift*), метода која улазне симболе пребацује на стек.
- Свођење (енг. *reduce*), метода која секвенцу симбола замјењује нетерминалом, примјеном неког продукционог правила.

Циљ је на крају да стек садржи само почетни симбол граматике, чиме се потврђује синтаксна исправност улаза. Ову технику користе сви LR парсери.

2.5.2.2. LR парсер

LR парсери представљају класу парсера за анализу навише који користе технику пребацивања и свођења. Назив LR означава:

- **L** слово означава читање улаза са лијева на десно (енг. ***L****eft-to-right*).
- **R** слово означава конструисање обрнуто крајње десно извођење стабла (енг. ***R****ightmost derivation in reverse*).

LR парсери доносе одлуке током анализе користећи стек и табелу анализе, на основу којих утврђују да ли треба наставити читање улаза, свести препознате симболе, прихватити улаз или пријавити синтаксну грешку.

2.6. Семантичка анализа

Семантичка анализа је завршна фаза етапе анализе у процесу компајлирања. Док синтаксна анализа провјерава структуру кода према граматици, семантичка анализа провјерава логичку исправност програма. Њен циљ је да осигура да су сви идентификатори исправно дефинисани, да се користе у складу са својим типовима, и да се поштују правила као што су видљивост и права приступа.

Неке од најчешћих семантичких грешака:

- Неслагање типова (енг. *type mismatch*).
- Коришћење недекларисаних промјенљивих
- Вишеструка декларација у истом опсегу.
- Приступ промјенљивој ван њеног опсега.
- Неслагање између формалних и стварних параматера функције.

Семантичка анализа се може изводити током или након синтаксне анализе. Једнопролазни компајлери врше синтаксну и семантичку анализу у једном пролазу, што је ефикасно, док вишепролазни компајлери одвајају ове кораке ради веће флексибилности и боље подршке за сложене структуре програма, овај принцип је имплементиран у Растију.

3. СПЕЦИФИКАЦИЈА СИСТЕМА

3.1. Функционалности система

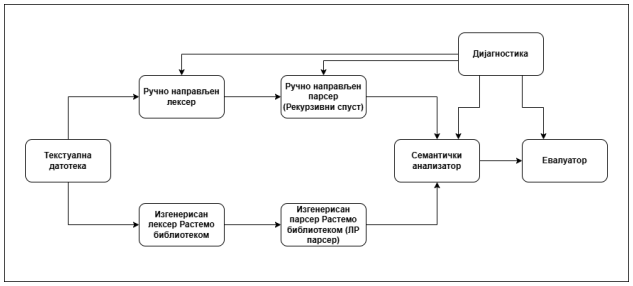
Пројекат Расти представља имплементацију компајлера написаног од нуле у програмском језику Раст. Подржава парсирање и евалуацију израза са нумеричким и буловим вриједностима, аритметичким, логичким, релационим и унарним операторима, као и рад са заградама. Поред тога, обухвата основне програмске конструкције као што су додјела, константе, условне наредбе, петље и угњеждени блокови кода.

3.2. Архитектура

Пројекат Расти представља имплементацију компајлера за императиван програмски језик. Иако не прати у потпуности класичне фазе компајлирања, састоји се од више компоненти, које су приказане на слици 2.

Архитектура приказана на слици обухвата:

- Ручно имплементиран лексер који учитава текст у радну меморију и генерише токене обрађујући га карактер по карактер.
- Ручно имплементиран парсер који гради синтаксно стабло методом рекурзивног спуста.
- Лексер изгенерисан Растемо библиотеком, који учитава токен по потреби и ради ефикасније од ручно имплементираног, јер не држи цијели код у радној меморији.
- LR парсер изгенерисан Растемо библиотеком.
- Семантички анализатор пролази кроз стабло парсирања и везује га семантичким правилама, при чему гради ново стабло.
- Након успјешне семантичке анализе, евалуатор обилази ново стабло, извршава наредбе, евалуира изразе и крајњи резултат приказује кориснику.
- Дијагностика која прикупља логове и грешке уколико се десе у свим фазама компајлирања.



Слика 2. *Архитектура пројекта*

3.3. Разлике између изгенерисаног и ручно имплементираног парсера

Највећа разлика између LR парсера и парсера са рекурзивним спустом лежи у приступу парсирању. LR парсер врши синтаксну анализу навише и обично се аутоматски генерише, док парсер са рекурзивним спустом обавља анализу наниже и пише се ручно. У овом поглављу биће разматране конкретне разлике између ових парсера имплементираних у пројекту.

Ручна имплементација парсера са рекурзивним спустом је једноставна и ефикасна за једноставне граматике, али мање проширива и осјетљива на граматичке проблеме као што је лијева рекурзија. Са друге стране, изгенерисани LR парсер пружа бољу подршку за проширења и аутоматску детекцију граматичких конфликата, што доприноси стабилности и лакшој одрживости пројекта.

Ефикасност је у оба случаја адекватна тренутној граматици, али LR приступ је прикладнији за даљи развој и сложеније граматике. Обје имплементације нуде сличан ниво дијагностике, коју чине информативне поруке и грешке.

4. ЗАКЉУЧАК

Рад представља приказ процеса изградње једноставног компајлера званог Расти, написаног у програмском језику *Rust* уз подршку библиотеке *Rustemo*. Основна мотивација за реализацију рада била је боље разумијевање процеса компајлирања и учење новог програмског језика.

На самом почетку рада дате су теоријске основе о томе шта је то компајлер, процес компајлирања и које разлике има у односу на интерпретере. Затим описана је етапа анализе коју чине лексичка, синтаксна и семантичка анализа. Посебан акценат стављен је на синтаксну анализу, гдје се обрађују методе синтаксне анализе наниже и навише.

У другом дијелу рада описана је спецификација система, која обухвата функционалност и архитектуру Растија. У оквиру тог поглавља дато је и образложење разлика између изгенерисаног и ручно имплементираног парсера.

Тренутна имплементација Растија пружа стабилну основу за будући развој и унапријеђење. Потенцијални правци даљег развоја укључују функционално проширење програмског језика, као што је увођење функција, структура или класа, што омогућава праћење савремених програмских стандарда. Такође, може се надоградити процес компајлирања, додавањем фаза као што су генерисање међукода, оптимизација и креирање циљног кода. Поред тога, простор за напредак јесте могућа имплементација неке од техника опоравка од грешака у лексичкој и синтаксној анализи, што доприноси робусности компајлера. Сви ови правци доприносе свеукупном квалитету компајлера и развојном потенцијалу програмског језика.

5. ЛИТЕРАТУРА

[1] Растемо (енг. *Rustemo*) библиотека - https://www.igordejanovic.net/rustemo/ (посљедњи приступ: 05.05.2025)

[2] Компајлери и интерпретери - https://www.spiceworks.com/tech/tech-general/articles/compiler-vs-interpreter-12-critical-differences-to-know/ (посљедњи приступ: 05.05.2025)

[3] Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D. *Compilers: Principles, Techniques, and Tools*. 2. издање. Boston: Addison-Wesley, 2006.

[4] Cooper, K. D., Torczon, L. *Engineering a Compiler*. 3. издање. Cambridge, MA: Morgan Kaufmann, 2022.

Кратка биографија:

Марко Бјелица рођен је у Требињу 1999. године. Мастер рад на Факултету техничких наука из области Рачунарство и аутоматика – Електронско пословање је одбранио 2025. године.

контакт: marko.bjelica19@gmail.com