**چکیده**

در این مقاله به بررسی طراحی و پیاده‌سازی یک کامپایلر ساده با زبان برنامه‌نویسی C می‌پردازیم. هدف اصلی از این پروژه، به‌کارگیری دانش نظری و مفاهیم آموزشی ارائه‌شده در کلاس های دانشگاه و بکارگیری آن‌ها در یک تجربه عملی است. این کامپایلر با هدف درک عمیق‌تر مفاهیمی همچون تحلیل لغوی[[1]](#footnote-1) ، تحلیل نحوی[[2]](#footnote-2) ، تحلیل معنایی[[3]](#footnote-3) و تولید کد نهایی، طراحی شده است و یک زبان برنامه‌نویسی ساده را کامپایل می‌کند. در این پروژه تلاش می‌کنیم با ارائه نمونه کدهای واقعی، دانشجویان را با فرآیندهای کلیدی در توسعه کامپایلر آشنا کنیم. همچنین، چالش‌ها و محدودیت‌های پروژه، مورد بحث قرار گرفته و مواردی برای بهینه سازی آن پیشنهاد شده است.

### ****بخش اول: مقدمه ای بر اجزای کامپایلر****

کامپایلر[[4]](#footnote-4) نرم افزاری برای تبدیل کد منبع [[5]](#footnote-5)به کد دودویی[[6]](#footnote-6) است. به عبارت دیگر می‌توان گفت که کامپایلر کدهای نوشته شده به زبان سطح بالا (نزدیک به زبان انسان) توسط برنامه نویسان را به زبان دودویی ماشین تبدیل می‌کند. (کامپیوترها تنها قادر به اجرای کدهای دودویی هستند) این فرآیند شامل مراحل مختلفی از جمله تحلیل لغوی ، تحلیل نحوی، تحلیل معنایی ، تولید کد میانی، بهینه‌سازی کد میانی و در نهایت تولید کد نهایی است.

**تحلیل‌گر لغوی:** بخشی از کامپایلر است که مستقیما به کد منبع دسترسی دارد. دنباله‌ای از کاراکترهای کد منبع را به صورت رشته‌ای از کاراکترها دریافت کرده، سپس این کاراکترها **توکن سازی** می‌شوند(لغات و نوع آنها تشخیص داده می‌شوند) و به عنوان جریانی از اطلاعات به مراحل بعدی یعنی تحلیل نحوی میرسند. این بخش خطاهای پایه‌ای مانند خطاهای نحوی را نیز شناسایی می‌کند.

**تحلیل‌گر نحوی:** وظیفه این بخش بررسی صحت و درستی ترتیب لغات برنامه مبدا است. این فرایند همچنین با نام پارسینگ[[7]](#footnote-7) نیز شناخته می‌شود. در طول این مرحله، کامپایلرها معمولا درخت‌های انتزاعی از سینتکس‌ها[[8]](#footnote-8) را ایجاد می‌کنند که ساختار‌های منطقی عناصر یک کد خاص را نشان می‌دهد. سپس کد یا عبارت مورد نظر را حرف به حرف مطابقت می‌دهد.

**تحلیل‌گر معنایی:** در این مرحله، **کامپایلر** کد‌ها را بر اساس منطق تعریف گشته بررسی می‌کند.

**تولید کد میانی:** این بخش از خروجی تحلیل‌گر معنایی، کد میانی را تولید میکند. هر کامپایلر می‌تواند کد میانی خاص خود را دارا باشد.

**بهینه‌سازی کد:** یکی دیگر از مراحل مهم در طراحی کامپایلر است که هدف آن تولید کدی با کیفیت بهتر و کارایی بالاتر است.

**تولید کد نهایی:** در نهایت، کد به زبان ماشین ترجمه می‌شود تا برنامه بر روی سخت‌افزار اجرا شود.

### بخش دوم: زبان طراحی‌شده و ویژگی های آن

### یک زبان بسیار ساده با هدف حذف پیچیدگی‌های غیرضروری به منظور طراحی و پیاده‌سازی یک کامپایلر ساخته شده است. ساختار این زبان اجرای وظایف ابتدایی و تحلیل آن‌ها را آسان می‌کند و با استفاده از عناصر پایه‌ای مانند دستورهای ورودی و خروجی، عملیات شرطی ساده، و برچسب‌ها، نمونه‌ای مناسب برای درک مفاهیم اولیه کامپایلر است. البته سادگی زیاد آن از محدودیت‌هایی مانند اجرای تنها یک عملگر در هر خط برخوردار می‌باشد که در ادامه به آن می‌پردازیم.

#### **ویژگی‌های زبان**

**۱.تعریف متغیرها:** زبان طراحی‌شده در این پروژه از ساختاری ساده و قابل فهم دارد. تعریف متغیرها در این زبان به سادگی انجام می‌شود و برای سادگی بیشتر تنها داده های عددی قابل تعریف هستند. کاربر می‌تواند متغیرها را مستقیماً مقداردهی کرده و در عملیات‌های مختلف از آن‌ها استفاده کند. این رویکرد باعث کاهش پیچیدگی نحوی شده و استفاده از زبان را آسان می‌کند.

**۲.عملگرها:** این زبان از مجموعه‌ای محدود اما کارآمد از عملگرها مانند چهار عمل اصلی پشتیبانی می‌کند. همچنین عملگر انتساب (=) برای مقداردهی به متغیرها و عملگرهای مقایسه‌ای مانند کوچک‌تر (<) و بزرگ‌تر (>) برای ارزیابی شرایط منطقی در اختیار کاربر قرار گرفته است. این مجموعه عملگرها، زبان را برای انجام محاسبات ریاضی ساده و پیاده‌سازی الگوریتم‌های پایه‌ای که به تصمیم‌گیری‌های شرطی نیاز دارند، کاملاً مناسب می‌کند.

**۳.ساختارهای کنترلی:** ساختارهای کنترلی زبان نیز به گونه‌ای طراحی شده‌اند که سادگی و کاربردی بودن را در اولویت قرار می‌دهند. شرط‌های ساده با استفاده از دستور if و کلمه کلیدی then تعریف می‌شوند و برای اجرای دستورات شرطی به صورت تک‌خطی کاربرد دارند. به علاوه، زبان از برچسب‌ها با استفاده از نشانه :label و دستورات پرش goto برای ایجاد حلقه‌ها پشتیبانی می‌کند. این رویکرد امکان کنترل جریان برنامه را به شکلی ساده اما مؤثر فراهم می‌کند.

**۴**.**ورودی و خروجی:** زبان طراحی‌شده همچنین شامل دستورات ورودی و خروجی است که به کاربر اجازه می‌دهد مقادیر را به برنامه وارد کرده و نتایج محاسبات را مشاهده کند. دستور input برای دریافت مقادیر از کاربر و دستور output برای نمایش مقادیر به خروجی استفاده می‌شود.

**۵.محدودیت‌ها:** در نهایت، این زبان با طراحی مینیمال خود محدودیت‌هایی را نیز اعمال کرده است که هدف آن ساده‌سازی فرآیند فهم و درک آن است. به عنوان مثال، هر خط از برنامه تنها می‌تواند شامل یک دستور یا عملیات باشد.

**fib\_seq = input**

**last = fib\_seq -1**

**i = 0**

**j = 1**

**n = 0**

**if n < 1 then goto :start**

**:comma**

**output ,**

**if n > 0 then goto :loop**

**:start**

**output i**

**output ,**

**output j**

**output ,**

**:loop**

**k = i + j**

**output k**

**i = j**

**j = k**

**n = n + 1**

**if n < last then goto :comma**

#### **نمونه کد ۱ تولید دنباله فیبوناچی**

به طور کلی، این زبان یک نمونه کاربردی و مؤثر از طراحی زبان‌های برنامه‌نویسی مینیمال(MiniLang) است که درک مفاهیم کلیدی مرتبط با کامپایلرها را به‌طور عملی ممکن می‌سازد.

### بخش سوم: تحلیل‌گر ‌لغوی

تحلیل لغوی یکی از مراحل ابتدایی در طراحی و پیاده‌سازی کامپایلر است که نقش مهمی در درک ساختار کد برنامه ایفا می‌کند. این مرحله، کد منبع را که به‌صورت متن ساده نوشته شده است، به اجزای کوچک‌تر و معنایی به نام توکن‌ها[[9]](#footnote-9) تجزیه می‌کند. هر توکن نماینده یک عنصر زبانی مانند کلیدواژه‌ها، عملگرها، شناسگرها، اعداد و نمادهای خاص است. در کد ارائه‌شده، ماژول تحلیل‌گر لغوی[[10]](#footnote-10) برای شناسایی این توکن‌ها از یک ساختار ساده و کارآمد استفاده می‌کند. توابع اصلی این بخش شامل خواندن کاراکتر از ورودی، شناسایی و جداسازی توکن‌ها، و مدیریت خطاها در صورت مواجهه با کاراکترهای نامعتبر است.

### عملکرد تحلیل لغوی در کد:

**۱.مدیریت ورودی و خروجی:** کاراکترهای کد منبع از طریق توابع lexer\_peek\_char و lexer\_read\_char مدیریت می‌شوند، که هرکدام نقش خاصی در فرآیند تحلیل لغوی ایفا می‌کنند.

**۱.۱. تابع lexer\_peek\_char:** برای مشاهده کاراکتر بعدی در رشته ورودی طراحی شده است بدون اینکه موقعیت فعلی در ورودی تغییر کند. این عملکرد امکان بررسی پیشاپیش را فراهم می‌کند، که برای تصمیم‌گیری درباره نوع توکن بعدی بسیار حیاتی است. به عنوان مثال، در هنگام شناسایی برچسب‌ها ، نیاز به بررسی کاراکترهای بعدی وجود دارد تا بتوان به‌طور دقیق نوع توکن را تعیین کرد.

**static char lexer\_peek\_char(struct lexer \*l)**

**{**

**if (l->read\_pos >= l->buffer\_len)**

**{**

**return EOF;**

**}**

**return l->buffer[l->read\_pos];**

**}**

#### lexer\_peek\_char **نمونه کد ۲ تابع**

**۲.۱. تابع lexer\_read\_char:** برای حرکت به کاراکتر بعدی استفاده می‌شود و هم‌زمان موقعیت فعلی در متن را به‌روزرسانی می‌کند. این تابع نه تنها کاراکتر جاری را به کامپایلر معرفی می‌کند بلکه وضعیت تحلیل‌گر لغوی را نیز به‌روز نگه می‌دارد، از جمله موقعیت خوانده‌شده و کاراکتر جاری.

**static char lexer\_read\_char(struct lexer \*l)**

**{**

**l->ch = lexer\_peek\_char(l);**

**l->pos = l->read\_pos;**

**l->read\_pos++;**

**return l->ch;**

**}**

**نمونه کد ۳ تابع lexer\_read\_char**

هماهنگی میان تابع peek و تابع read برای مدیریت جریان متن و جلوگیری از خواندن مجدد یا از دست دادن کاراکترها بسیار مهم است.

**۲.حذف فاصله‌ها:** با استفاده از تابع skip\_whitespaces، فاصله‌های خالی مانند فاصله، تب، و خط‌جدید که در تحلیل بی‌اهمیت هستند، نادیده گرفته می‌شوند.

**static void skip\_whitespaces(struct lexer \*l)**

**{**

**while (isspace(l->ch))**

**{**

**lexer\_read\_char(l);**

**}**

**}**

**نمونه کد ۴ تابع skip\_whitespaces**

**۳.شناسایی انواع توکن‌ها:** هر توکن بر اساس نوع و محتوای آن شناسایی می‌شود:

**۳.۱.کلیدواژه‌ها و شناسگرها:** با بررسی نام‌ها و مطابقت آن‌ها با لیست کلیدواژه‌ها (مانند input، if، و goto ) نوع توکن مشخص می‌شود. توکن‌هایی که کلیدواژه نیستند به‌عنوان شناسگر (identifier) در نظر گرفته می‌شوند.

**۳.۲.اعداد صحیح:** توکن‌های عددی با بررسی توالی ارقام شناسایی و به‌عنوان عدد (integer) ذخیره می‌شوند.

**۳.۳.عملگرها و نمادهای خاص:** عملگرها مانند (+) ، (-) و نمادهایی مانند( : ) و( , ) مستقیماً شناسایی می‌شوند.

**۳.۴.برچسب‌ها:** رشته‌هایی که با ( : ) شروع می‌شوند به‌عنوان برچسب شناسایی می‌شوند.

**۳.۵.خطاها:** در صورت وجود کاراکتر نامعتبر، توکن به‌عنوان نامعتبر (invalid) برچسب‌گذاری می‌شود.

**if (strcmp(value, "if") == 0)**

**{**

**return (struct token){.kind = TK\_IF, .value = NULL};**

**}**

**نمونه کد ۵ تابع شناسایی توکن if**

**۴.ذخیره و ارائه توکن‌ها:** توکن‌ها با استفاده از ساختار struct token ذخیره شده و در آرایه‌ای پویا نگهداری می‌شوند. این آرایه نماینده لیستی از تمام اجزای کد منبع است که در مراحل بعدی توسط تحلیل نحوی و معنایی استفاده خواهد شد.

**int lexer\_tokenize(char \*buffer, unsigned int length, ds\_dynamic\_array \*tokens)**

**{**

**struct lexer lexer;**

**lexer\_init(&lexer, buffer, length);**

**struct token tok;**

**do**

**{**

**tok = lexer\_next\_token(&lexer);**

**if (ds\_dynamic\_array\_append(tokens, &tok) != 0)**

**{**

**DS\_PANIC("Failed to append token to array");**

**}**

**} while (tok.kind != END);**

**return 0;**

**}**

**نمونه کد ۶ تابع تبدیل به توکن و ذخیره سازی در آرایه**

### بخش چهارم: تحلیل‌گر ‌نحوی

تحلیل نحوی برای تحلیل گرامر زبان ورودی استفاده می‌شود. پارسر ساخته شده در کامپایلر تحلیل گرامر را از طریق شناسایی توکن‌ها و ساخت گرههای دستوری (مانند عبارات شرطی ، برچسب‌ها و..) انجام می‌دهد. به طور کلی تحلیل نحوی کامپایلر به شرح زیر است:

**۱. دریافت و خواندن توکن‌ها:** توکن‌های تولیدی توسط تحلیل‌گر لغوی را دریافت و با کمک توابع parser\_init ، parser\_current و parser\_advance پیمایش و پردازش می‌کند.

**void parser\_init(ds\_dynamic\_array tokens, struct parser \*p)**

**{**

**p->tokens = tokens;**

**p->index = 0;**

**}**

**void parser\_current(struct parser \*p, struct token \*tok)**

**{**

**ds\_dynamic\_array\_get(&p->tokens, p->index, tok);**

**}**

**void parser\_advance(struct parser \*p)**

**{**

**p->index++;**

**}**

**نمونه کد ۷ توابع خواندن و دریافت توکن‌ها**

**۲. شناسایی دستورها :** هر توکن توسط تابع parse\_instr بررسی می‌شود تا نوع دستور تعیین گردد:

* اگر توکن یک متغیر باشد (TK\_IDENT) ، دستور به‌عنوان تخصیص (INSTR\_ASSIGN) تحلیل می‌شود.
* اگر توکن if باشد، دستور شرطی (INSTR\_IF) تحلیل شده و عبارات منطقی و دستورهای داخلی آن پردازش می‌شوند.
* اگر توکن goto باشد، دستور انتقال (INSTR\_GOTO) با شناسایی برچسب مقصد تحلیل می‌شود.
* اگر توکن output باشد، دستور چاپ (INSTR\_OUTPUT) تحلیل شده و متغیر چاپ‌شونده مشخص می‌شود.
* برچسب‌ها (TK\_LABEL) و سایر دستورها نیز به همین ترتیب با استفاده از توابع مشخص شناسایی می‌شوند.

**void parse\_instr(struct parser \*p, struct instr\_node \*instr)**

**{**

**struct token token;**

**parser\_current(p, &token);**

**if (token.kind == TK\_IDENT)**

**{**

**parse\_assign(p, instr);**

**}**

**else if (token.kind == TK\_IF)**

**{**

**parse\_if(p, instr);**

**}**

**else if (token.kind == TK\_GOTO)**

**{**

**parse\_goto(p, instr);**

**}**

**...**

**else**

**{**

**DS\_PANIC("unexpected token %s", show\_token\_kind(token.kind));**

**}**

**{**

**نمونه کد ۸ تابع تشخیص دستورات کامپایلر**

**۳.ساخت درخت نحو:** درخت نحو یک ساختار داده درختی است که نمایش دهنده تحلیل نحوی کد ورودی است. هر گره از این درخت نماینده یک بخش از دستور زبان برنامه‌نویسی یا توکن‌های آن است.مراحل ساخت آن به عبارت‌اند از:

**۱.۳. تحلیل توکن‌ها:** توکن‌های ورودی (که توسط lexerتولیده شده‌اند) یک‌به‌یک پردازش شده تا ساختار دستوری آن‌ها مشخص شود.

**۲.۳. ایجاد گره برای هر نوع دستور:** هر دستور یک نوع گره مختص به خود دارد.

**۳.۳. اتصال گره‌ها برای ساخت سلسله مراتب:** گره‌های مربوط به عبارت‌ها به گره اصلی دستور متصل می‌شوند.

**INSTR\_IF**

**/ | \**

**REL\_NODE THEN INSTR\_GOTO**

**/ \**

**TERM TERM**

**n 3**

نمونه ساختار درخت نحو عبارت if n < 3 then goto :loop

### بخش پنجم: تحلیل‌گر ‌معنایی

### تحلیل معنایی برای بررسی صحت برنامه و اطمینان از انطباق با قوانین معنایی زبان برنامه‌نویسی انجام می‌شود. این مرحله، پس از تحلیل نحوی، به عنوان گامی کلیدی در کامپایلر عمل می‌کند که منطق برنامه را بررسی می‌کند. تحلیل معنایی، توسط توابعی مانند parse\_expr ، **parse\_program ، parser\_if و ... انجام میشود.**

**۱. بررسی منطق عبارات :** تابع parse\_expr در پردازش دستورات، وظیفه شناسایی عبارات را برعهده دارد. عبارات شامل عملیات ریاضی، منطقی، یا مقادیر منفرد است. این تابع توکن‌های ورودی و ساختار‌های عبارات را بررسی کرده .

**۱.۱. بررسی توکن اولیه**:ابتدا یک عبارت بااصطلاح آغاز می‌شود. این مقدار می‌تواند یک عدد (Integer)، شناسه (Identifier)، یا ورودی کاربر (Input) باشد.(تابع parse\_term برای این منظور فراخوانی می‌شود)

**۲.۱. بررسی عملیات ریاضی:** پس از شناسایی ترم اولیه، توکن‌های بعدی بررسی می‌شوند تا مشخص شود آیا یک عملگر ریاضی (مانند جمع +، تفریق – و ...) وجود دارد یا خیر. به ازای هر یک از این عملگرها:

* گره جدیدی در ساختار expr\_node ساخته می‌شود.
* ترم سمت راست (Right-hand side) از عبارت نیز با فراخوانی مجدد parse\_term پردازش می‌شود.

**۳.۱. بازگشت برای مقادیر اصطلاح:** اگر هیچ عملگر ریاضی شناسایی نشود، عبارت به عنوان یک اصطلاح در نظر گرفته می‌شود.

**void parse\_expr(struct parser \*p, struct expr\_node \*expr)**

**{**

**struct token token;**

**struct term\_node lhs, rhs;**

**parse\_term(p, &lhs);**

**parser\_current(p, &token);**

**if (token.kind == TK\_PLUS)**

**{**

**parser\_advance(p);**

**parse\_term(p, &rhs);**

**expr->kind = EXPR\_PLUS;**

**expr->add.lhs = lhs;**

**expr->add.rhs = rhs;**

**}**

**else if (token.kind == TK\_MINUS)**

**{**

**parser\_advance(p);**

**parse\_term(p, &rhs);**

**expr->kind = EXPR\_MINUS;**

**expr->min.lhs = lhs;**

**expr->min.rhs = rhs;**

**}**

**{**

**نمونه کد ۱۰ تابع تشخیص و بررسی منطق عبارات کامپایلر**

**۲. بررسی منطق دستورات:** با استفاده از توابعی مانند parse\_if ، parse\_goto ، parse\_label و... برای بررسی منطق دستورات استفاده می‌شود.

**void parse\_if(struct parser \*p, struct instr\_node \*instr)**

**{**

**struct token token;**

**instr->kind = INSTR\_IF;**

**parser\_advance(p);**

**parse\_rel(p, &instr->if\_term.rel);**

**parser\_current(p, &token);**

**if (token.kind != TK\_THEN)**

**{**

**DS\_PANIC("Expected Then found %s", show\_token\_kind(token.kind));**

**}**

**parser\_advance(p);**

**instr->if\_term.instr = malloc(sizeof(struct instr\_node));**

**parse\_instr(p, instr->if\_term.instr);**

**}**

**نمونه کد ۱۱ تابع بررسی منطق دستور شرطی if** در **کامپایلر**

**۳. اتصال تمامی دستورات:** پس از بررسی دستورات و عبارات توسط توابع مربوطه، تمامی دستورات با استفاده از تابع parse\_program به یک دیگر متصل و برای تولید کد میانی آماده می‌شوند.

**void parse\_program(struct parser \*p, struct program\_node \*program)**

**{**

**ds\_dynamic\_array\_init(&program->instrs, sizeof(struct instr\_node));**

**struct token token;**

**do**

**{**

**struct instr\_node instr;**

**parse\_instr(p, &instr);**

**ds\_dynamic\_array\_append(&program->instrs, &instr);**

**parser\_current(p, &token);**

**} while (token.kind != END);**

**}**

**نمونه کد ۱۲ تابع اتصال دستورات کامپایلر**

### بخش ششم: تولید کد میانی

1. Lexical Analysis [↑](#footnote-ref-1)
2. Syntax Analysis [↑](#footnote-ref-2)
3. Semantic Analysis [↑](#footnote-ref-3)
4. Compiler [↑](#footnote-ref-4)
5. Source Code [↑](#footnote-ref-5)
6. ‌Binary Code [↑](#footnote-ref-6)
7. Parsing [↑](#footnote-ref-7)
8. Syntax Tree (AST) [↑](#footnote-ref-8)
9. Tokens [↑](#footnote-ref-9)
10. Lexer [↑](#footnote-ref-10)