Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

Wydział Informatyki

Projekt dyplomowy

Implementacja narzędzi lex i yacc z wykorzystaniem metaprogramowania

Implementation of lexical analyzer (lex) and parser generator (yacc) tools using metaprogramming techniques

Autorzy: Bartosz Buczek, Bartłomiej Kozak
Kierunek studiów: Informatyka
Opiekun pracy: dr inż. Tomasz Służalec

Kraków, 2025

Spis treści

Rozdział 1

Cel prac i wizja projektu

1.1. Charakterystyka problemu

Leksery i parsery są kluczowymi elementami w procesie tworzenia interpreterów i kompilatorów języków programowania. Pozwalają one przekształcić kod źródłowy napisany przez programistę na reprezentację wewnętrzną, wykorzystywaną później przez dalsze etapy przetwarzania kodu.

Analiza leksykalna wykonywana przez lekser polega na rozdzieleniu kodu źródłowego na jednostki logiczne, zwane leksemami. Parser natomiast wykonuje analizę składniową w celu ustalenia struktury gramatycznej tekstu i jej zgodności z gramatyką języka.

Celem pracy inżynierskiej jest stworzenie narzędzia *ALPACA* (Another Lexer Parser And Compiler Alpaca) w języku Scala, które implementuje funkcjonalności powszechnie stosowane w budowie lekserów i parserów.

1.2. Motywacja projektu

Projekt ma na celu stworzenie nowoczesnego narzędzia do generowania lekserów i parserów w języku Scala, łączącego zalety istniejących rozwiązań z nowoczesnym podejściem technologicznym. Jego główne cele to:

1. Stworzenie intuicyjnego API.

2. Opracowanie obszernej dokumentacji.

3. Rozbudowana diagnostyka błędów.

4. Poprawa wydajności względem rozwiązań w języku Python.

5. Integracja z popularnymi środowiskami programistycznymi (IDE).

Proponowane rozwiązanie łączy nowoczesne podejście technologiczne z praktycznym zastosowaniem w edukacji i programowaniu. Może on służyć jako narzędzie dydaktyczne, ułatwiając naukę teorii kompilacji, w pracach badawczych, a także jako kompleksowe narzędzie do tworzenia praktycznych rozwiązań.

1.3. Przegląd istniejących rozwiązań

Dostępne na rynku rozwiązania umożliwiają tworzenie analizatorów, jednak charakteryzują się ograniczeniami związanymi z wydajnością, wysokim progiem wejścia i diagnostyką błędów.

1.3.1. Lex, Yacc

Lex[**lesk1975lex**] i *Yacc*[**johnson1975yacc**] to klasyczne, dobrze ugruntowane narzędzia, które odegrały kluczową rolę w tworzeniu setek współczesnych języków programowania. Definicja leksera i parsera w tych systemach odbywa się poprzez specjalnie zaprojektowaną składnię konfiguracyjną. Mimo pewnych zalet, jego złożoność i wysoki próg wejścia mogą stanowić wyzwanie.

Ponieważ *Lex* i *Yacc* zostały zaprojektowane do współpracy z językiem C, ich integracja z nowoczesnymi językami programowania bywa utrudniona. Rozszerzanie tych narzędzi o dodatkowe, specyficzne funkcjonalności jest skomplikowane, co ogranicza ich elastyczność. Brak wsparcia dla współczesnych środowisk programistycznych (IDE) dodatkowo obniża komfort użytkowania w porównaniu z nowoczesnymi alternatywami.

```
{
/*%%%*/
value_expr($3);
$1->nd_value = $3;
$$ = $1;
/*%
$$ = dispatch2(massign, $1, $3);
%*/
}
| var_lhs tOP_ASGN command_call
{
value_expr($3);
$$ = new_op_assign($1, $2, $3);
}
| primary_value '[' opt_call_args rbracket tOP_ASGN command_call
{
/*%%%*/
NODE *args;

value_expr($6);
if (!$3) $3 = NEW_ZARRAY();
args = arg_concat($3, $6);
if ($5 == tOROP) {
    $5 = 0;
}
else if ($5 == tANDOP) {
    $5 = 1;
}
$$ = NEW_OP_ASGN1($1, $5, args);
fixpos($$, $1);
/*%
$$ = dispatch2(aref_field, $1, escape_Qundef($3));
$$ = dispatch3(opassign, $$, $5, $6);
%*/
}
```

Listing 1.1: Fragment definicji parsera Ruby w technologii Yacc

1.3.2. PLY, SLY

PLY[**ply**] i jego nowszy odpowiednik *SLY*[**sly**] to biblioteki inspirowane narzędziami Lex i Yacc. Oferują elastyczne podejście do budowy parserów, umożliwiając samodzielną implementację obsługi leksemów, budowę drzewa AST, czy dodatkowe funkcjonalności takie jak obliczanie numeru linii w lekserze.

Głównym ograniczeniem PLY i SLY jest implementacja w języku Python. Ze względu na interpretowany charakter oraz dynamiczne typowanie, parsery te charakteryzują się niską wydajnością, a brak statycznego typowania utrudnia wykrywanie błędów na etapie kompilacji. Przy implementacji parserów z użyciem biblioteki SLY w środowisku PyCharm obserwuje się wiele ostrzeżeń dotyczących potencjalnych naruszeń reguł, co często wymaga zastosowania mechanizmów supresji, aby uniknąć fałszywie pozytywnych wyników analizy statycznej kodu. Ponadto należy zaznaczyć, iż autor projektu informuje o braku dalszego rozwoju tych narzędzi[**sly-github**].

Przykład 1.2 ilustruje kilka nieintuicyjnych, automatycznych mechanizmów obecnych w bibliotece *SLY*.

- Operator `@_()` jest zdefiniowany, aby automatycznie analizować tekst przy pomocy wyrażeń regularnych. Literały muszą być zawarte w cudzysłowie, a „zmienna” odpowiada za matchowany „typ”.
- Nazwa metody oznacza „typ” zwracany przez daną produkcję, czyli dla definicji `IF` należy najpierw odszukać wszystkie metody, które mają nazwę `condition`, gdyż są to możliwe produkcje.
- W krotce (sic!) `precedence` definiujemy pierwszeństwo operatorów, jednakże dodanie `% prec` pozwala nadpisać priorytet dla konkretnej reguły składniowej.
- Argument `p` pozwala na dostęp do kontekstu produkcji (np. numeru linii), ale także do zmiennych w patternu match w adnotacji. Jeśli zdefiniowany jest więcej niż jeden, to dodajemy numer do accesora, np. `expr1` jest odwołaniem się do drugiego wyrażenie `expr`. Jednocześnie, można to zrobić także poprzez odwołanie się do konkretnego indeksu obiektu `p`.

```
class MatrixParser(Parser):
tokens = MatrixScanner.tokens

precedence = (
    ('nonassoc', 'IFX'),
    ('nonassoc', 'ELSE'),
    ('nonassoc', 'EQUAL'),
)

@_('"{" instructions "}"')
def block(self, p: YaccProduction):
    raise NotImplementedError

@_('instruction')
```

```
def block(self, p: YaccProduction):
    raise NotImplementedError

@_('IF "(" condition ")" block %prec IFX')
def instruction(self, p: YaccProduction):
    raise NotImplementedError

@_('IF "(" condition ")" block ELSE block')
def instruction(self, p: YaccProduction):
    raise NotImplementedError

@_('expr EQUAL expr')
def condition(self, p: YaccProduction):
    args = [p.expr0, p.expr1]
    raise NotImplementedError
```

Listing 1.2: Fragment definicji parsera w Pythonie, wykorzystujacy bibliotekę SLY

Komunikaty błędów w bibliotece *SLY* są bardzo ograniczone, co obrazuje przykład 1.3, który po uruchomieniu informuje użytkownika błędem z fragmentu kodu **??**. Okazuje się, że problemem był brak atrybutu `ignore_comment` w definicji `Lexer`.

```
tokens = Scanner().tokenize("a = 1 + 2")
for tok in tokens:
    print(tok)
```

Listing 1.3: Fragment niedziałajacego kodu w Pythonie, wykorzystujacy bibliotekę SLY

```
  File "main.py", line 2, in <module>
    for tok in tokens:
               ^^^^^^
  File "Python\site-packages\sly\lex.py", line 374, in tokenize
    _set_state(type(self))
    ~~~~~~~~~~^^^^^^^^^^^^
  File "Python\site-packages\sly\lex.py", line 367, in _set_state
    _master_re = cls._master_re
                 ^^^^^^^^^^^^^^
AttributeError: type object 'Scanner' has no attribute '_master_re'
```

Listing 1.4: Przykładowy komunikat błędu w bibliotece *SLY*

1.3.3. ANTLR

ANTLR[**parr2004s**] to kolejne rozwiązanie inspirowane narzędziami *Lex* i *Yacc*, oferujące zaawansowane mechanizmy analizy składniowej. Jego twórcy opracowali dedykowany język DSL, znany jako Grammar v4, który umożliwia definiowanie składni analizowanego języka. Na podstawie tej definicji *ANTLR* generuje parser w wybranym przez użytkownika języku programowania, takim jak Python, Java, C++ lub JavaScript.

Wspomaganie pracy z *ANTLR* w znacznym stopniu ułatwiają dedykowane wtyczki do środowisk Visual Studio Code oraz IntelliJ IDEA. Oferują one funkcjonalności, takie jak kolorowanie składni, autouzupełnianie kodu, nawigację do definicji leksemów oraz walidację błędów, co znacząco przyspiesza proces tworzenia parserów.

Jedną z kluczowych różnic *ANTLR* w porównaniu do innych narzędzi jest wykorzystanie gramatyki LL(*), podczas gdy klasyczne rozwiązania, takie jak Yacc czy SLY,

implementują LALR(1). LL(*) jest bardziej intuicyjna i czytelna dla programistów, co ułatwia definiowanie reguł składniowych. Jednakże, jej zastosowanie wiąże się z większym zużyciem pamięci oraz niższą wydajnością w porównaniu do LALR(1).

Dodatkowym wyzwaniem podczas korzystania z *ANTLR* jest konieczność nauki składni DSL Grammar v4 oraz ograniczenie wsparcia dla narzędzi deweloperskich. Pełne wykorzystanie możliwości *ANTLR* wymaga korzystania z jednego z dedykowanych środowisk, co może stanowić istotne ograniczenie dla użytkowników preferujących inne IDE.

1.3.4. Scala parser combinators

Biblioteka *Scala parser combinators*[**moors2008parser**] była popularnym sposobem na tworzenie parserów, lecz jak wynika z dokumentacji, „Trudno jest jednak zrozumieć ich działanie i jak zacząć. Po skompilowaniu i uruchomieniu kilku pierwszych przykładów, mechanizm działania staje się bardziej zrozumiały, ale do tego czasu może to być zniechęcające, a standardowa dokumentacja nie jest zbyt pomocna"[**parser-combinators-readme**].

1.3.5. ScalaBison

Z podsumowania artykułu na temat *ScalaBison*[**boyland2010tool**] wiadomo, że to praktyczny generator parserów dla języka Scala oparty na technologii rekurencyjnego wstępowania i zstępowania, który akceptuje pliki wejściowe w formacie *bison*. Parsery generowane przez *ScalaBison* używają bardziej informacyjnych komunikatów o błędach niż te generowane przez pierwowzór *bison*, a także szybkość parsowania i wykorzystanie miejsca są znacznie lepsze niż *scala-combinators*, ale są nieco wolniejsze niż najszybsze generatory parserów oparte na JVM.

Dodatkowo należy zaznaczyć, iż jest to rozwiązanie już niewspierane i stworzone w celach akademickich. Korzysta z przestarzałej wersji Scali, nie posiada wyczerpującej dokumentacji i liczba funkcjonalności jest bardzo ograniczona w porównaniu do np. technologii *SLY*.

1.3.6. parboiled2

parboiled2[**myltsev2019parboiled2**] to biblioteka w Scali umożliwiająca lekkie i szybkie parsowanie dowolnego tekstu wejściowego. Implementuje ona oparty na makrach generator parsera dla gramatyk wyrażeń parsujących (PEG), który działa w czasie kompilacji i tłumaczy definicję reguły gramatycznej na odpowiadający jej bytecode JVM. Niestety próg wejścia ze względu na skomplikowany i nieintuicyjny DSL jest wysoki. Zgodnie z przykładem 1.5, raportowanie błędów jest bardzo ograniczone (problem z implementacją wynika jedynie z różnic w liczbie parametrów funkcji).

```
1 [error] /Users/haoyi/Dropbox (Personal)/Workspace/scala-js-book/scalatexApi
     /src/main/scala/scalatex/stages/Parser.scala:60: overloaded
2 method value apply with alternatives:
3 [error] [I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, RR](f: (I, J
     , K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, scalatex.stages.Ast.
     Block.
4 Text, scalatex.stages.Ast.Chain, Int, scalatex.stages.Ast.Block) => RR)(
     implicit j: org.parboiled2.support.ActionOps.SJoin[shapeless.::[I,
```

```
shapeless.::[J,shapeless.::[K,shapeless.::[L,shapeless.::[M,shapeless.::[N,
    shapeless.::[O,shapeless.::[P,shapeless.::[Q,shapeless.::[R,
shapeless.::[S,shapeless.::[T,shapeless.::[U,shapeless.::[V,shapeless.::[W,
    shapeless.::[X,shapeless.::[Y,shapeless.::[Z,shapeless.
HNil]]]]]]]]]]]]]]]]]]],shapeless.HNil,RR], implicit c: org.parboiled2.
    support.FCapture[(I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z,
    scalatex.
stages.Ast.Block.Text, scalatex.stages.Ast.Chain, Int, scalatex.stages.Ast.
    Block) => RR])org.parboiled2.Rule[j.In,j.Out] <and>
[error] [J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, RR](f: (J, K, L
    , M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, scalatex.stages.Ast.Block.
    Text,
scalatex.stages.Ast.Chain, Int, scalatex.stages.Ast.Block) => RR)(implicit
    j: org.parboiled2.support.ActionOps.SJoin[shapeless.::[J,
shapeless.::[K,shapeless.::[L,shapeless.::[M,shapeless.::[N,shapeless.::[O,
    shapeless.::[P,shapeless.::[Q,shapeless.::[R,shapeless.::[S,
shapeless.::[T,shapeless.::[U,shapeless.::[V,shapeless.::[W,shapeless.::[X,
    shapeless.::[Y,shapeless.::[Z,shapeless.HNil]]]]]]]]]]]]]]]]]],shapeless.
    HNil,RR], implicit c: org.parboiled2.support.FCapture[(J, K, L, M, N, O,
     P, Q, R, S,
T, U, V, W, X, Y, Z, scalatex.stages.Ast.Block.Text, scalatex.stages.Ast.
    Chain, Int, scalatex.stages.Ast.Block) => RR])org.parboiled2.Rule[j.
In,j.Out] <and>
```

Listing 1.5: Niewielki fragment (14 z 133 linii) błędu wygenerowanego przez bibliotekę *parboiled2*, który pochodzi z prezentacji Li Haoyi na temat *FastParse*[**fastparse-talk**].

1.3.7. FastParse

FastParse*FastParse*[**fastparse-docs**] to opracowana przez Li Haoyi, wysokowydajna biblioteka kombinatorów parserów dla Scali, zaprojektowana w celu uproszczenia tworzenia parserów tekstu strukturalnego. Umożliwia ona programistom definiowanie parserów rekurencyjnych, dzięki czemu nadaje się do parsowania języków programowania, formatów danych, takich jak JSON, czy DSL-i. Cechą charakterystyczną FastParse jest równowaga między użytecznością a wydajnością. Parsery są konstruowane poprzez łączenie mniejszych parserów za pomocą operatorów, takich jak **~** dla sekwencjonowania i **|** dla alternatyw, przy jednoczesnym zachowaniu czytelności zbliżonej do formalnych definicji gramatyki. Według dokumentacji[**fastparse-docs**], parsery *Fastparse* zajmują 1/10 kodu w porównaniu do ręcznie napisanego parsera rekurencyjnego. W porównaniu do narzędzi generujących parsery, takich jak *ANTLR* lub *Lex* i *Yacc*, implementacja nie wymaga żadnego specjalnego kroku kompilacji lub generowania kodu. To sprawia, że rozpoczęcie pracy z *Fastparse* jest znacznie łatwiejsze niż w przypadku bardziej tradycyjnych narzędzi do generowania parserów. Przykładowo, parser wyrażeń arytmetycznych może być zwięźle napisany, aby obsługiwać zagnieżdżone nawiasy, pierwszeństwo operatorów i raportowanie błędów w mniej niż 20 liniach kodu[**fastparse-slides**]. Biblioteka kładzie również nacisk na debugowanie, generując szczegółowe komunikaty o błędach, które wskazują dokładną lokalizację i przyczynę niepowodzeń parsowania, takich jak niedopasowane nawiasy lub nieprawidłowe tokeny.

1.3.8. Podsumowanie

Narzędzie	**Lex&Yacc**	**PLY/SLY**	**ANTLR**	**scala-bison**
Język implementacji	C	Python	Java	Scala (nad Bisonem)
Język użycia	regex, BNF, akcje w C	DSL	DSL oparty na EBNF	BNF, akcje w Scali
Wydajność	wysoka	niska	umiarkowana	wysoka
Łatwość użycia	średnia	umiarkowana	wysoka	średnia
Aktywne wsparcie	brak	nie	tak	nie
Diagnostyka błędów	słaba	średnia	dobra	słaba
Dokumentacja	dobra	średnia, nieaktualna	dobra	słaba
Popularność	wysoka	średnia	wysoka	niska
Integracja IDE	nieoficjalny plugin	ograniczona	oficjalny plugin	brak
Wsparcie do debugowania	brak	dobre	częściowe	dobre
Generowania kodu	nie	nie	tak	nie

Narzędzie	**Scala parser combinators**	**parboiled2**	**FastParse**	**ALPACA**
Język implementacji	Scala	Scala	Scala	Scala
Język użycia	DSL w Scali	DSL w Scali	DSL w Scali	Scala
Wydajność	wysoka	umiarkowana	wysoka	TODO
Łatwość użycia	niska	średnia	średnia	TODO
Aktywne wsparcie	nie	nie	tak	TODO
Diagnostyka błędów	dobra	niska	dobra	TODO
Dokumentacja	słaba	bardzo dobra	bardzo dobra	TODO
Popularność	średnia	niska	rosnąca	TODO
Integracja IDE	wsparcie dla Scali	wsparcie dla Scali	wsparcie dla Scali	TODO
Wsparcie do debugowania	dobre	dobre	dobre	TODO
Generowania kodu	nie	nie	nie	TODO

Tabela 1.1: Porównanie wybranych narzędzi do generowania lekserów i parserów

Rozdział 2

Metaprogramowanie w Scali 3

2.1. Wprowadzenie

Scala 3, znana również jako Dotty, wprowadza całkowicie przeprojektowany system metaprogramowania, stanowiący fundamentalną zmianę w stosunku do eksperymentalnych makr dostępnych w Scali 2[**scala3-dropped-scala2-macros**, **scala3-metaprogramming**].

Metaprogramowanie w Scali 3 zostało zaprojektowane z naciskiem na bezpieczeństwo typów, przenośność oraz skalowalność, oferując programistom możliwość generowania i analizowania kodu w czasie kompilacji przy zachowaniu pełnej ekspresywności języka[**stucki2024infoscience**, **stucki2020inlining**]. W przeciwieństwie do poprzedniego systemu, który eksponował wewnętrzne mechanizmy kompilatora i był źródłem problemów z kompatybilnością między wersjami[**stucki2020thesis**], nowy system metaprogramowania jest zaprojektowany jako stabilny i przenośny interfejs programistyczny. Podstawą teoretyczną systemu metaprogramowania w Scali 3 jest programowanie wieloetapowe (multi-stage programming), paradygmat pozwalający na odróżnienie różnych etapów wykonania programu[**scala3-staging**, **stucki2020thesis**]. W tym modelu kod może być wykonywany w różnych fazach: w czasie kompilacji (compile-time), w czasie wykonania (runtime)[**scala3-staging**].

2.1.1. Quotes i splices

Kluczowymi koncepcjami w systemie metaprogramowania Scali 3 są quotes i splices[**stucki2018unification**, **stucki2021multistage**]. Quotes, oznaczane jako `'{...}`, służą do opóźnienia wykonania kodu i traktowania go jako danych[**scala3-reflection**, **epfl-dotty-reflection**]. Splices, oznaczane jako `${...}`, pozwalają na ocenę wyrażenia generującego kod i wstawienie wyniku do otaczającego kontekstu[**scala3-reflection**, **epfl-dotty-reflection**, **scala3-guides-quotes**].

Formalna semantyka tych konstrukcji została przedstawiona w pracy Stuckiego, Brachthäusera i Odersky'ego[**stucki2021multistage**], gdzie quotes i splices są traktowane jako prymitywne formy w typowanych drzewach składni abstrakcyjnej (typed abstract syntax trees). Autorzy dowodzą, że system zachowuje bezpieczeństwo typów oraz higieniczność, zapewniając, że wygenerowany kod nie może przypadkowo powiązać identyfikatorów z niewłaściwymi zmiennymi[**stucki2021multistage**].

2.1.2. Bezpieczeństwo międzyetapowe

Scala 3 gwarantuje bezpieczeństwo międzyetapowe (cross-stage safety) poprzez sprawdzanie poziomów etapowania w czasie kompilacji[**stucki2020thesis**, **stucki2021multistage**]. Zmienne lokalne mogą być używane tylko na tym samym poziomie etapowania, na którym zostały zdefiniowane, co zapobiega dostępowi do zmiennych, które jeszcze nie istnieją lub już nie są dostępne[**stucki2020thesis**].

System również zapewnia, że typy generyczne używane w wyższym poziomie etapowania niż ich definicja wymagają instancji klasy typu `Type[T]`, która niesie reprezentację typu niepoddaną wymazywaniu (type erasure)[**stucki2020thesis**]. To podejście rozwiązuje problem wymazywania typów generycznych w JVM, zachowując informację o typach potrzebną w kolejnych etapach kompilacji.

2.2. Mechanizmy metaprogramowania w Scali 3

2.2.1. Definicje inline

Najprostszym narzędziem metaprogramowania jest modyfikator `inline`. Gwarantuje on, że wywołanie oznaczonej nim metody lub wartości zostanie w całości **wstawione w miejscu wywołania** (ang. *inlining*) podczas kompilacji. Jest to polecenie dla kompilatora, a nie tylko sugestia, jak w niektórych innych językach

2.2.2. Makra oparte na wyrażeniach

Makra w Scali 3 są zdefiniowane jako metody `inline` zawierające splice najwyższego poziomu (top-level splice)[**scala3-reference-macros**, **scala3-guides-macros**], czyli taki, który nie jest zagnieżdżony w żadnym quotes i jest wykonywany w czasie kompilacji[**scala3-staging**, **scala3-reference-macros**].

Typ `Expr[T]` reprezentuje wyrażenie Scali o typie `T` jako typowane drzewo składni abstrakcyjnej[**scala3-reflection**, **scala3-guides-macros**]. Makra manipulują wartościami typu `Expr[T]`, transformując je lub generując nowe wyrażenia[**scala3-guides-macros**]. Ta reprezentacja gwarantuje bezpieczeństwo typów na poziomie języka metaprogramowania[**scala3-reflection**].

2.2.3. Refleksja TASTy

Dla przypadków wymagających głębszej analizy kodu, Scala 3 oferuje API refleksji TASTy (Typed Abstract Syntax Tree)[**scala3-reflection**, **epfl-dotty-reflection**, **scala3-guides-reflection**]. TASTy jest binarnym formatem serializacji typowanych drzew składni abstrakcyjnej używanym przez kompilator Scali 3[**stucki2020thesis**].

API refleksji dostarcza szczegółowy widok na strukturę kodu, włączając typy, symbole oraz pozycje w kodzie źródłowym[**scala3-reflection**, **scala3-guides-reflection**]. Jest dostępne poprzez obiekt `reflect` zdefiniowany w typie `Quotes`, który jest przekazywany kontekstualnie do makr[**scala3-reflection**, **scala3-guides-reflection**].

2.3. Implementacja systemu metaprogramowania

2.3.1. Architektura kompilatora

Implementacja systemu metaprogramowania w Scali 3 jest zorganizowana wokół kilku kluczowych komponentów[**stucki2020thesis**]. Proces kompilacji obejmuje fazę inliningu (*Inlining* phase), która rozwija definicje `inline` oraz wywołania makr[**dotty-compiler-phases**].

Faza *PostInlining* wykonuje czyszczenie po rozwinięciu definicji inline, usuwając pomocnicze struktury i optymalizując kod[**dotty-compiler-phases**]. Faza *Staging* zajmuje się sprawdzaniem poziomów etapowania oraz adaptacją typów etapowanych poprzez proces zwany "type healing"[**stucki2020thesis**, **dotty-compiler-phases**].

2.3.2. Dopasowanie wzorców w cytatach

Scala 3 wspiera analizę kodu poprzez dopasowanie wzorców w quotes (*quote pattern matching*)[**stucki2020thesis**, **stucki2021multistage**]. Mechanizm ten pozwala na dekonstrukcję kawałków kodu i ekstrakcję podwyrażeń[**stucki2021multistage**].

Stucki, Brachthäuser i Odersky[**stucki2021multistage**] wprowadzają wzorce wiążące (*bind patterns*) postaci `$x` oraz wzorce HOAS (Higher-Order Abstract Syntax) postaci `$f(y)`, które pozwalają na ekstrakcję podwyrażeń potencjalnie zawierających zmienne z zewnętrznego kontekstu[**stucki2021multistage**]. System gwarantuje, że ekstrahowane wyrażenia są zamknięte względem definicji wewnątrz wzorca, zapobiegając wyciekom zakresu[**stucki2021multistage**].

Rozdział 3

Implementacja

3.1. Praktyczna implementacja analizatora leksykalnego z wykorzystaniem makr w Scali 3

3.1.1. Wprowadzenie do studium przypadku

Niniejszy rozdział prezentuje praktyczną implementację systemu analizy leksykalnej (leksera) wykorzystującego zaawansowane mechanizmy metaprogramowania Scali 3. Przedstawiony kod stanowi przykład zastosowania technik opisanych w poprzednich rozdziałach do rozwiązania rzeczywistego problemu inżynierskiego: automatycznej generacji wydajnego analizatora leksykalnego z definicji wysokopoziomowej w formie języka dziedzinowego (DSL).

System `alpaca.lexer` implementuje transformację deklaratywnych reguł tokenizacji zapisanych jako funkcja częściowa (partial function) w kod procedualny wykonywany w czasie kompilacji. Wykorzystuje przy tym pełne spektrum możliwości refleksji TASTy, włączając generację klas w czasie kompilacji, transformację drzew AST oraz wyspecjalizowane typy refinement.

3.1.2. Architektura systemu leksera

Interfejs użytkownika

System oferuje użytkownikowi przejrzysty interfejs DSL oparty na dopasowaniu wzorców:

```
type LexerDefinition[Ctx <: AnyGlobalCtx] = PartialFunction[String,
    Token[?, Ctx, ?]]
```

Listing 3.1: Definicja typu LexerDefinition

Definicja `LexerDefinition` reprezentuje reguły leksera jako funkcję częściową mapującą wzorce wyrażeń regularnych (jako ciągi znaków) na definicje tokenów. Wykorzystanie funkcji częściowej pozwala na naturalne wyrażenie reguł leksykalnych w idiomatycznej składni Scali.

Główny punkt wejścia systemu stanowi metoda `inline`:

```
transparent inline def lexer[Ctx <: AnyGlobalCtx & Product](
  using Ctx WithDefault DefaultGlobalCtx,
```

```
)(
  inline rules: Ctx ?=> LexerDefinition[Ctx],
)(using
  copy: Copyable[Ctx],
  betweenStages: BetweenStages[Ctx],
): Tokenization[Ctx] =
  ${ lexerImpl[Ctx]('{ rules }, '{ summon }, '{ summon }) }
```

Listing 3.2: Punkt wejścia: transparent inline def lexer

Modyfikator `transparent inline` zapewnia, że zwracany typ będzie dokładnie odpowiadał wygenerowanej strukturze, włączając typy refinement dla poszczególnych tokenów. Użycie parametrów kontekstowych (`using`) realizuje wzorzec dependency injection na poziomie systemu typów.

Implementacja makra

Implementacja makra `lexerImpl` stanowi serce systemu:

```
private def lexerImpl[Ctx <: AnyGlobalCtx: Type](
  rules: Expr[Ctx ?=> LexerDefinition[Ctx]],
  copy: Expr[Copyable[Ctx]],
  betweenStages: Expr[BetweenStages[Ctx]],
)(using quotes: Quotes,
): Expr[Tokenization[Ctx]] = {
```

Listing 3.3: Implementacja makra: podpis lexerImpl

Makro przyjmuje wyrażenie reprezentujące reguły leksera jako `Expr[Ctx ?⇒ LexerDefinition[Ctx]]` oraz instancje kontekstualnych klas pomocniczych. Parametr `using Quotes` dostarcza dostępu do API refleksji TASTy.

3.1.3. Analiza drzewa składni abstrakcyjnej

Dekonstrukcja funkcji częściowej

Kluczowym krokiem implementacji jest ekstrakcja reguł z definicji funkcji częściowej:

```
  val Lambda(oldCtx :: Nil, Lambda(_, Match(_, cases: List[CaseDef]))) =
    rules.asTerm.underlying.runtimeChecked
```

Listing 3.4: Dekonstrukcja funkcji częściowej (dopasowanie AST do CaseDef)

Ten fragment kodu wykorzystuje dopasowanie wzorców w cytatach (quote pattern matching) do dekonstrukcji typowanego AST funkcji częściowej. Struktura `Lambda(_, Match(_, cases))` odpowiada wewnętrznej reprezentacji funkcji częściowej, gdzie `Match` zawiera listę przypadków `CaseDef`.

Metoda `asTerm` konwertuje `Expr[T]` na `Term`, umożliwiając bezpośrednią manipulację AST. Pole `underlying` dostarcza dostęp do wewnętrznej reprezentacji, a `runtimeChecked` zapewnia walidację w czasie wykonania makra.

Przetwarzanie przypadków dopasowania

Każdy przypadek (`CaseDef`) jest przetwarzany indywidualnie w operacji fold:

```
1  val (tokens, infos) = cases.foldLeft((tokens =
    List.empty[Expr[ThisToken]], infos = List.empty[TokenInfo[?]])):
2    case ((tokens, infos), CaseDef(tree, None, body)) =>
```

Listing 3.5: Fold po przypadkach dopasowania do zbudowania listy tokenów i metadanych

Struktura `CaseDef(tree, None, body)` dekonstruuje przypadek dopasowania na wzorzec (`tree`), opcjonalny guard (`None` w tym przypadku) oraz ciało (`body`). System obecnie nie wspiera przypadków z guardami, co jest jawnie zasygnalizowane w kodzie.

3.1.4. Transformacja i adaptacja referencji

Klasa replacerefs

Kluczową techniką jest zastąpienie referencji do starego kontekstu nowymi referencjami:

```
1     def replaceWithNewCtx(newCtx: Term) = new
   ReplaceRefs[quotes.type].apply(
2       (find = oldCtx.symbol, replace = newCtx),
3       (find = tree.symbol, replace = Select.unique(newCtx,
   "lastRawMatched")),
4     )
```

Listing 3.6: Zastąpienie referencji starego kontekstu nowymi (ReplaceRefs)

Transformacja ta realizuje proces znany jako ”re-owning” w terminologii kompilatorów — zmianę właściciela (owner) symboli w AST. Jest to konieczne, ponieważ kod oryginalnie odnoszący się do parametru makra musi zostać przepisany, aby odnosił się do parametru metody w wygenerowanej klasie.

Konstrukcja `Select.unique(newCtx, "lastRawMatched")` generuje wyrażenie dostępu do pola obiektu, odpowiadające składni `newCtx.lastRawMatched` w kodzie Scali.

Funkcja CreateLambda

Klasa `CreateLambda` służy do generowania wyrażeń lambda w AST:

```
1            val ctxManipulation = createLambda[CtxManipulation[Ctx]] {
   case (methSym, (newCtx: Term) :: Nil) =>
2              replaceWithNewCtx(newCtx).transformTerm(
3                Block(statements.map(_.changeOwner(methSym)),
   Literal(UnitConstant())),
4              )(methSym)
5            }
```

Listing 3.7: Generowanie lambdy z re-owningiem ownerów

Metoda `changeOwner` modyfikuje symbol właściciela dla wszystkich węzłów w podrzewie AST. Jest to kluczowe dla zapewnienia, że wygenerowany kod będzie poprawnie typowany i możliwy do wykonania.

3.1.5. Ekstrakcja i kompilacja wzorców

Funkcja extractSimple

Funkcja `extractSimple` implementuje logikę dopasowania różnych typów definicji tokenów:

```
1       def extractSimple(ctxManipulation: Expr[CtxManipulation[Ctx]])
2         : PartialFunction[Expr[ThisToken], List[(Expr[ThisToken],
    Expr[TokenInfo[?]])]] =
3         case '{ Token.Ignored(using $ctx) } =>
4           compileNameAndPattern[Nothing](tree).map { case '{ $tokenInfo:
    TokenInfo[name] } =>
5             '{ IgnoredToken[name, Ctx]($tokenInfo, $ctxManipulation) } ->
    tokenInfo
6           }
7         case '{ type t <: ValidName; Token.apply[t]($value: v)(using $ctx)
    } =>
8           compileNameAndPattern[t](tree).map { case '{ $tokenInfo:
    TokenInfo[name] } =>
9             // we need to widen here to avoid weird types
10            TypeRepr.of[v].widen.asType match
11              case '[result] =>
12                val remapping = createLambda[Ctx => result] { case
    (methSym, (newCtx: Term) :: Nil) =>
13
    replaceWithNewCtx(newCtx).transformTerm(value.asTerm)(methSym)
14              }
15              '{ DefinedToken[name, Ctx, result]($tokenInfo,
    $ctxManipulation, $remapping) } -> tokenInfo
16          }
```

Listing 3.8: Funkcja extractSimple: dopasowywanie definicji tokenów

Wykorzystuje ona dopasowanie wzorców w cytatach z ekstraktorem typów, umożliwiając rozróżnienie różnych wariantów definicji tokenów na poziomie typów. Konstrukcja `type t <: ValidName` w wzorcu wiąże parametr typu do zmiennej wzorca `t`, umożliwiając jego późniejsze wykorzystanie.

compilenameandpattern

Pomocnicza klasa `CompileNameAndPattern` jest odpowiedzialna za walidację i kompilację wzorców wyrażeń regularnych:

3.2. Analiza wzorców: klasa CompileNameAndPattern

3.2.1. Wprowadzenie

Klasa `CompileNameAndPattern` stanowi kluczowy komponent systemu analizy leksykalnej, odpowiedzialny za ekstrakcję i walidację wzorców tokenów podczas ekspansji makra. Jej głównym zadaniem jest transformacja różnorodnych form wzorców występujących w definicjach DSL na ujednolicone struktury `TokenInfo`, które następnie są wykorzystywane do generacji finalnego kodu leksera.

Implementacja wykorzystuje rekurencyjne przetwarzanie drzewa AST z zastosowaniem optymalizacji rekurencji ogonowej (`@tailrec`), co zapewnia efektywność działania nawet dla złożonych wzorców z wieloma alternatywami.

3.2.2. Architektura klasy

Klasa jest sparametryzowana typem `Quotes`, co pozwala na pełną integrację z API refleksji TASTy:

```
1 private[lexer] final class CompileNameAndPattern[Q <: Quotes](using val
     quotes: Q) {
2   import quotes.reflect.*
3   def apply[T: Type](pattern: Tree): List[Expr[TokenInfo[?]]] =
```

Listing 3.9: Architektura klasy CompileNameAndPattern: sygnatura i pola

Parametr typu `T: Type` reprezentuje typ wzorca w systemie typów Scali 3, który jest analizowany w celu określenia nazwy tokena. Wzorzec `pattern: Tree` to fragment drzewa AST reprezentujący lewą stronę przypadku dopasowania w definicji leksera.

3.2.3. Algorytm rekurencyjnego przetwarzania

Główna metoda `apply` deleguje przetwarzanie do wewnętrznej funkcji rekurencyjnej `loop`:

Funkcja `loop` przyjmuje:

- `tpe: TypeRepr` — reprezentację typu wzorca, wykorzystywaną do określenia nazwy tokena
- `pattern: Tree` — drzewo AST wzorca, z którego ekstrahowany jest wzorzec wyrażenia regularnego

Adnotacja `@tailrec` zapewnia, że kompilator zweryfikuje poprawność optymalizacji rekurencji ogonowej, co jest kluczowe dla wydajności przy przetwarzaniu złożonych wzorców.

3.2.4. Przypadki dopasowania wzorców

Wzorzec z wiązaniem i literałem

Pierwszy przypadek obsługuje sytuację, gdy nazwa tokena jest wyprowadzana z nazwy zmiennej wiązania:

```
1         case (TermRef(qual, name), Bind(bind,
     Literal(StringConstant(regex)))) if name == bind =>
2           Result.unsafe(regex, regex) :: Nil
```

Listing 3.10: Wiązanie i literał: wyprowadzenie nazwy tokena

Struktura AST: Ten przypadek odpowiada wzorcowi postaci:

```
1 case x @ "\\d+" => Token[x.type]
```

gdzie:

- `TermRef(qual, name)` — reprezentacja typu tokena jako referencja do termu
- `Bind(bind, ...)` — operacja wiązania zmiennej `x`
- `Literal(StringConstant(regex))` — literał ciągu znaków zawierający wyrażenie regularne
- warunek `name == bind` — weryfikacja, że nazwa typu odpowiada nazwie zmiennej

Rezultat: nazwa tokena i wzorzec są identyczne (oba równe `regex`).

Wzorzec z wiązaniem i alternatywami

Drugi przypadek rozszerza poprzedni o obsługę alternatyw:

```
      case (TermRef(qual, name), Bind(bind, Alternatives(alternatives))) if name == bind =>
        alternatives
          .map {
            case Literal(StringConstant(str)) => Result.unsafe(str, str)
            case x => raiseShouldNeverBeCalled(x.show)
          }
```

Listing 3.11: Wiązanie z alternatywami: generacja wielu TokenInfo

Struktura AST: Odpowiada wzorcowi:

```
case x @ ("+" | "-" | "*" | "/") => Token[x.type]
```

gdzie `Alternatives(alternatives)` reprezentuje konstrukcję `("+ ...)`. System generuje osobny `TokenInfo` dla każdej alternatywy, wszystkie z tą samą nazwą tokena (wyprowadzoną z `x`).

Wzorzec ignorowany z literałem

Trzeci przypadek obsługuje tokeny ignorowane (bez przypisanej nazwy):

```
      case (tpe, Literal(StringConstant(str))) if tpe =:= TypeRepr.of[Nothing] =>
        Result.unsafe(str, str) :: Nil
```

Listing 3.12: Token ignorowany: literał regex

Struktura AST: Odpowiada wzorcowi:

```
case "\\s+" => Token.Ignored
```

Typ `Nothing` sygnalizuje, że token nie ma przypisanej nazwy w systemie typów. Zarówno nazwa jak i wzorzec są ustawiane na wartość literału.

Wzorzec ignorowany z alternatywami

Czwarty przypadek łączy tokeny ignorowane z alternatywami:

```
      case (tpe, Alternatives(alternatives)) if tpe =:= TypeRepr.of[Nothing] =>
        alternatives.map {
          case Literal(StringConstant(str)) => Result.unsafe(str, str)
          case x => raiseShouldNeverBeCalled(x.show)
        }
```

Listing 3.13: Token ignorowany: alternatywy regex

Struktura AST: Odpowiada wzorcowi:

```
case "\\s+" | "\\t+" | "\\n+" => Token.Ignored
```

Wzorzec z typem stałym i literałem

Piąty przypadek obsługuje jawne określenie nazwy tokena przez typ:

```
      case (ConstantType(StringConstant(name)), Literal(StringConstant(regex))) =>
        Result.unsafe(name, regex) :: Nil
```

Listing 3.14: Typ stały i literał: jawna nazwa tokena

Struktura AST: Odpowiada wzorcowi:

```
case "\\d+" => Token["number"]
```

gdzie `ConstantType(StringConstant(name))` reprezentuje typ literalny ciągu znaków `"number"`. W tym przypadku nazwa tokena (`name`) i wzorzec (`regex`) są rozdzielone.

Wzorzec z typem stałym i alternatywami

Szósty przypadek łączy jawną nazwę z alternatywnymi wzorcami:

```
      case (ConstantType(StringConstant(str)), Alternatives(alternatives)) =>
        Result.unsafe(
          str,
          alternatives
            .map {
              case Literal(StringConstant(str)) => str
              case x => raiseShouldNeverBeCalled(x.show)
            }
            .mkString("|"),
        ) :: Nil
```

Listing 3.15: Typ stały i alternatywy: łączenie wzorców

Struktura AST: Odpowiada wzorcowi:

```
case ("+" | "-") => Token["operator"]
```

Wszystkie alternatywy są łączone w jeden wzorzec z operatorem `|`, a nazwa tokena jest wspólna dla wszystkich.

Wzorzec z zagnieżdżonym wiązaniem

3.2.5. Walidacja i konstrukcja TokenInfo

Walidacja nazwy tokena

Funkcja `validateName` zapewnia, że nazwy tokenów spełniają wymagania systemu:

```
private def validateName(name: String)(using quotes: Quotes): ValidName =
  import quotes.reflect.*
  name match
    case invalid @ "_" =>
      report.errorAndAbort(s"Invalid token name: $invalid")
    case other => other
```

Nazwa `"_"` jest zarezerwowana w Scali i nie może być użyta jako nazwa tokena. W przypadku wykrycia niepoprawnej nazwy, kompilacja jest przerywana z informacyjnym komunikatem błędu.

Konstrukcja wyrażenia TokenInfo

Obiekt `Result` zawiera metodę `unsafe`, która konstruuje wyrażenie `TokenInfo`:

```
def unsafe(name: String, regex: String)(using quotes: Quotes):
    Expr[TokenInfo[?]] = {
  import quotes.reflect.*
  val validatedName = validateName(name)
  ConstantType(StringConstant(validatedName)).asType match
    case '[type nameTpe <: ValidName; nameTpe] =>
      '{ TokenInfo[nameTpe](
           ${ Expr(validatedName).asExprOf[nameTpe] },
           ${ Expr(regex) }
         ) }
}
```

Proces konstrukcji obejmuje:

1. **Walidację nazwy:** wywołanie `validateName(name)`
2. **Konwersję na typ:** `ConstantType(StringConstant(validatedName)).asType`
3. **Wiązanie typu:** dopasowanie wzorcem typu `[type nameTpe <: ValidName; nameTpe]`
4. **Konstrukcję wyrażenia:** cytowanie kodu tworzącego `TokenInfo`