

Kompilator języka strukturalnego

David Korenchuk

August, 23, 2023

Spis treści

1	Wprowadzenie	3
2	Historia	4
3	Teoria	5
3.1	Języki formalne	5
3.2	Klasyfikacja gramatyczna	5
4	Analiza leksykalna	6
4.1	Wyrażenia regularne	6
4.2	Flex	7
5	Analiza składniowa	8
5.1	Definicja	8
5.2	Eliminacja rekurencji lewej	9
5.3	Niejednoznaczność	10
5.4	Implementacja AST	11
5.5	Implementacja analizatora składniowego	11
5.6	Reprezentacja wizualna AST	11
6	Analiza semantyczna	13
6.1	Analiza nieużywanych zmiennych	13
7	System typów	14
7.1	Opis	14
7.2	Definicja systemu	14
8	Generacja kodu pośredniego	16
9	Optymalizacje kodu pośredniego	17
9.1	Constant propagation	17
10	Interpreter	18
11	Annex: Gramatyka w BNF	19

1 Wprowadzenie

Człowiek posługuje się językami werbalnymi, aby komunikować z innymi ludźmi. Za pomocą języka polskiego albo angielskiego można wyrazić myśl, ale zazwyczaj w sposób niejednoznaczny, bo jesteśmy przyzwyczajeni do tego, że każde zdanie może być wyrażone na wiele sposobów. Natomiast, aby umożliwić komunikację pomiędzy człowiekiem a komputerem, te zdania muszą być dość mocno sprecyzowane, aby móc je wykonać w sposób deterministyczny.

Celem niniejszej pracy jest pokazanie technik, które są używane do umożliwiania takiego rodzaju komunikacji. Dalsza część pracy zawiera opis każdego z etapów tworzenia języka programowania strukturalnego.

2 Historia

Potrzeba automatyzacji pracy intelektualnej istniała zawsze. Dlatego od dawna człowiek próbuje znaleźć metody do tego. Niżej jest krótkie podsumowanie powstania informatyki.

- W **IX** wieku przez irańskiego matematyka al Kindi został stworzony system szyfrowania informacji na podstawie zliczania ilości liter w tekście.
- W **XVII** wieku powstał suwak logarytmiczny, potrzebny do ułatwienia działań matematycznych.
- W tym samym **XVII** wieku powstał jeden z pierwszych kalkulatorów mechanicznych **Pascalina**. Jest to narzędzie do wykonania operacji arytmetycznych na podstawie ruchu koł zębatych i innych części.
- W **XVIII** wieku Charlesa Babbage stworzył mechaniczną **maszynę różnicową** do tworzenia dużych tabeli logarytmicznych, które do tej pory człowiek musiał wyliczać ręcznie.
- W 1847 roku George Boole wyprowadził nowy rozdział algebry: **algebrę Boole’a**, na podstawie której później został zaprojektowany pierwszy klasyczny komputer.
- W 1930 roku Vannevar Bush stworzył **analizator różnicowy** do rozwiązywania równań różnicowych metodą całkowania.

3 Teoria

3.1 Języki formalne

Według teorii automatów, automat – jest to jednostka wykonawcza. Jednostki te, zależnie od swojej struktury i tego, jaki **język formalny** oni mogą obrobić, dzielą się na klasy.

Klasy te opisane są **hierarchią Chomsky’ego**. Mówi ona o tym, że języki formalne dzielą się na 4 typy:

- Typ 3 – języki regularne
- Typ 2 – języki bezkontekstowe
- Typ 1 – języki kontekstowe
- Typ 0 – języki rekurencyjnie przeliczalne

Jako przykład języka typu 3 według hierarchii Chomsky’ego można podać wyrażenia regularne. Język ten opisuje się automatem skończonym deterministycznym (DFA). Bardziej szczegółowo wyrażenia regularne będą rozpatrzone w opisanu analizy leksykalnej.

3.2 Klasyfikacja gramatyczna

Niniejszy język nie może być odniesiony do żadnej z klas hierarchii Chomsky’ego, chociaż jest on językiem regularnym. Tak jest dlatego, że można napisać gramatycznie poprawny kod, który jednak prowadzi do błędów kontekstowych i logicznych. Naprzykład

```
void f() {  
    return argument + 1;  
}
```

Kolejną z przyczyn niemożliwości odniesienia naszego języka do jednej z klas hierarchii Chomsky’ego jest niejednoznaczność konstrukcji językowych. Przykład niżej pokazuje, że nie można jednoznacznie stwierdzić, czy `data * d` jest deklaracją zmiennej albo operatorem mnożenia dwóch zmiennych. Aby móc poprawnie prowadzić analizę składniową, musimy zadbać o rozróżnienie kontekstu.

```
void f() {  
    data *d;  
}
```

4 Analiza leksykalna

Jednym ze sposobów na sprowadzanie kodu źródłowego do postaci listy tokenów jest narzędzie flex. Przyjmuje ono zestaw reguł w postaci wyrażeń regularnych, według których działa rozbięcie tekstu wejściowego. Można jednak ominąć lex i zaimplementować lexer ręcznie, ale ta praca nie skupia się na tym.

4.1 Wyrażenia regularne

Wyrażenie regularne – łańcuch znaków, zawierający pewne polecenia do wyszukiwania tekstu.

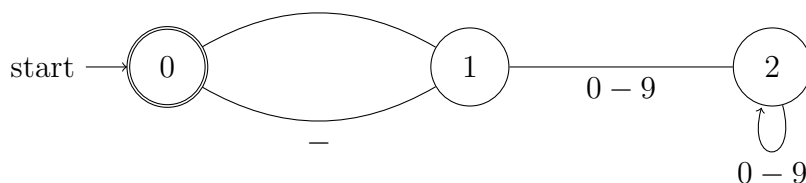
Mówimy, że wyrażenie regularne określone nad alfabetem Σ , jeżeli zachodzą następujące warunki:

- \emptyset – wyrażenie regularne, reprezentujące pusty zbiór.
- ϵ – wyrażenie regularne, reprezentujące pusty łańcuch.
- $\forall_{a \in \Sigma}$, a reprezentuje jeden znak.
- Warunek indukcyjny: jeżeli R_1, R_2 – wyrażenia regularne, $(R_1 R_2)$ stanowi konkatencję R_1 i R_2 .
- Warunek indukcyjny: jeżeli R – wyrażenie regularne, R^* stanowi domknięcie Kleene'ego.

W rzeczywistości, takich zasad może być więcej.

Zazwyczaj wyrażenie regularne jest realizowane za pomocą DFA (Deterministic finite automaton, Deterministyczny automat skończony). Lex sprowadza podany zbiór zasad do takiego automatu.

Podamy przykład automatu dla wyrażenia $-?[0-9]^+$



Aby odśledzić wykonane kroki, można wypełnić tabelę przejść pomiędzy stanami. Podamy przykład dla łańcucha -22

Bieżący stan	Akcja
0	zaakceptować -
0, 1	zaakceptować 2
0, 1, 2	zaakceptować 2

4.2 Flex

Flex jest narzędziem projektu GNU. Pozwala ono w wygodny sposób podać listę reguł dla analizy leksykalnej (ang. Scanning). Flex jest mocno powiązany z językiem C, dlatego program w flex'u korzysta z konstrukcji języka C. Pokażemy przykład użycia flex'u

Listing 1: Przykład użycia flex

```
%{
#include "portrzebny-do-analizy-plik.h"

/* Kod w języku C. */
%}

/* Opcje flex */
%option noyywrap nounput noinput
%option yylineno

%% /* Reguły w postaci wyrażen regularnych. */

/*****
/* Wzorzec          | Akcja przy znalezieniu takiego wzorca */
*****/
-?[0-9]+             LEX_CONSUME_WORD(TOK_INTEGRAL_LITERAL)
-?[0-9]+\.[0-9]+     LEX_CONSUME_WORD(TOK_FLOATING_POINT_LITERAL)
\"([^\\"\\\\]*(\\\"\\\\\"\\\\\"*))\" LEX_CONSUME_WORD(TOK_STRING_LITERAL)
\\'\\.\\'             LEX_CONSUME_WORD(TOK_CHAR_LITERAL)

.                   { /* Znaleziony niewiadomy znak.
                      Zglosic blad.
                      */ }

%%
```

Zauważmy, że flex próbuje szukać wzorców w tekście dokładnie w takiej kolejności, która jest podana w jego kodzie. Dlatego często robią ostatnią regułę z wyrażeniem regularnym ".", który akceptuje dowolny znak, i umieszczają tam komunikat o błędzie.

W naszym przypadku, lex generuje kod, który gromadzi wszystkie znalezione lexemy do tablicy.

5 Analiza składniowa

5.1 Definicja

Mając listę składników elementarnych wejściowego programu, jesteśmy w stanie przejść do następnego etapu kompilacji – analizy składniowej. Jest to proces generacji struktury drzewiastej, a mianowicie AST (Abstract Syntax Tree).

Wynikiem działania analizy składniowej zawsze jest **jedno** drzewo AST. Może zawierać ono definicje wszystkich funkcji.

AST może być stworzony po zdefiniowaniu gramatyki regularnej danego języka. Stosuje się do tego notacja BNF (Backus–Naur form). Pełny opis gramatyki pokazany jest w końcu pracy. Pokażemy tylko kilka przykładów:

$$\begin{aligned}
 \langle program \rangle & ::= (\langle function-decl \rangle \mid \langle structure-decl \rangle)^* \\
 \langle var-decl \rangle & ::= \langle type \rangle (*)^* \langle id \rangle = \langle logical-or-stmt \rangle ; \\
 \langle stmt \rangle & ::= \langle block-stmt \rangle \\
 & \quad \mid \langle selection-stmt \rangle \\
 & \quad \mid \langle iteration-stmt \rangle \\
 & \quad \mid \langle jump-stmt \rangle \\
 & \quad \mid \langle decl \rangle \\
 & \quad \mid \langle expr \rangle \\
 & \quad \mid \langle assignment-stmt \rangle \\
 & \quad \mid \langle primary-stmt \rangle
 \end{aligned}$$

W przypadku wyrażeń arytmetycznych, AST także jednoznacznie określa za pomocą produkcji gramatyki BNF priorytet operacji arytmetycznych. Na przykład, mając wyrażenie $1 + 2 * 3 + 4$, drzewo syntaktyczne będzie skonstruowane zgodnie z prawami arytmetyki, co pozwala nie trzymać w AST żadnych informacji o nawiasach. Widać, że aby zastosować produkcję $\langle additive-stmt \rangle$, najpierw musi być zastosowana następna produkcja $\langle multiplicative-stmt \rangle$.

Pomocnicza przy prowadzeniu analizy jest **tablica parsingu**. Jest to zbiór konkretnych przejść pomiędzy produkcjami. Pomaga ona w zrozumieniu, jaką produkcję zastosować mając dany nieterminal. Zauważmy, że w tabelę są wpisane produkcje bez alternatyw, i każde przejście gramatyczne określone jednoznacznie.

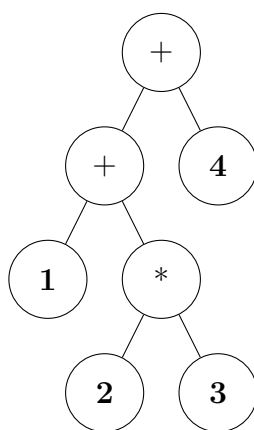
Aby zbudować tę tablicę, możemy użyć zasady **First & Follow**. Tutaj **First** to zbiór terminalnych symboli, które mogą pojawić się jako pierwsze w ciągu znaków wygenerowanym przez daną nieterminalną symbol w gramatyce, a **Follow** to zbiór terminalnych symboli, które mogą wystąpić bezpośrednio po danym nieterminalnym symbolu w dowolnym ciągu znaków wygenerowanym przez gramatykę.

Podamy gramatykę dla przykładu powyżej ($1 + 2 * 3 + 4$). Musimy wprowadzić dwa poziomy priorytety, aby prawidłowo zachować kolejność operacji mnożenia i dodawania.

$$\begin{aligned}
 \langle \text{additive-stmt} \rangle & ::= \langle \text{multiplicative-stmt} \rangle \\
 & \quad | \langle \text{multiplicative-stmt} \rangle + \langle \text{additive-stmt} \rangle \\
 & \quad | \langle \text{multiplicative-stmt} \rangle - \langle \text{additive-stmt} \rangle \\
 \langle \text{multiplicative-stmt} \rangle & ::= \langle \text{prefix-unary-stmt} \rangle \\
 & \quad | \langle \text{prefix-unary-stmt} \rangle * \langle \text{multiplicative-stmt} \rangle \\
 & \quad | \langle \text{prefix-unary-stmt} \rangle / \langle \text{multiplicative-stmt} \rangle \\
 & \quad | \langle \text{prefix-unary-stmt} \rangle \% \langle \text{multiplicative-stmt} \rangle
 \end{aligned}$$

	First	Follow
$\langle \text{additive-stmt} \rangle$	0-9	+, -
$\langle \text{multiplicative-stmt} \rangle$	0-9	*, /
$\langle \text{prefix-unary-stmt} \rangle$	0-9	€

	0-9	+	-	*	/	\$
$\langle \text{additive-stmt} \rangle$		$\langle \text{mul} \rangle +$ $\langle \text{add} \rangle$	$\langle \text{mul} \rangle -$ $\langle \text{add} \rangle$			$\langle \text{mul} \rangle$
$\langle \text{multiplicative-stmt} \rangle$				$\langle \text{una} \rangle *$ $\langle \text{mul} \rangle$	$\langle \text{una} \rangle /$ $\langle \text{mul} \rangle$	$\langle \text{una} \rangle$
$\langle \text{prefix-unary-stmt} \rangle$	0-9					



5.2 Eliminacja rekurencji lewej

Projektując gramatykę, należy wziąć pod uwagę problem rekurencji lewej (Left recursion). Są produkcje gramatyczne, nie pozwalające kodu, które je implementuje przejść do następnego terminalu, stosując tę

samą produkcję, co prowadzi do rekurencji nieskończonej.

Rekurencja lewa może wyglądać następująco:

$$\langle factor \rangle ::= \langle factor \rangle '+' \langle term \rangle$$

Kod, wykonujący tą regułę będzie miał postać:

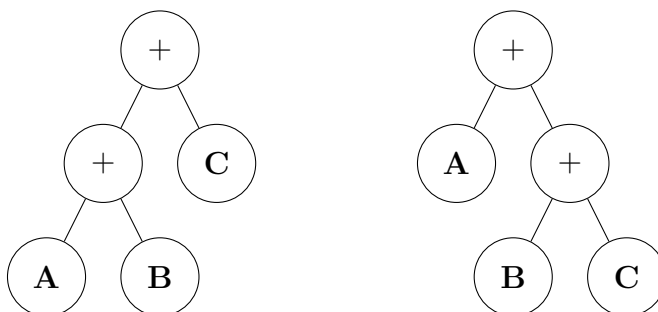
```
void factor() {
    factor(); // Rekurencja bez zadnego warunku wyjścia
    consume(' ');
    term();
}
```

5.3 Niejednoznaczność

Projektując język, łatwo trafić na niejednoznaczne produkcje gramatyczne. One są takie, że jednego tekstu wejściowego są one w stanie wyprodukować kilka różnych od siebie drzew. Popularny warunek dla stworzenia niejednoznaczności to taka produkcja

$$\begin{aligned} \langle P \rangle &::= \langle P \rangle + \langle P \rangle \\ &\quad | \langle symbol \rangle \end{aligned}$$

Po zastosowaniu danej produkcji dla $A + B + C$ możliwe jest otrzymanie dwóch drzew



Aby rozwiązać ten problem i jednoznacznie wskazać kolejność zastosowania reguł gramatycznych, możemy zamienić prawy operand na symbol, wtedy eliminuje się dwuznaczność. Pierwsza produkcja poniżej jest **lewostronną**, a druga – **prawostronną**.

$$\langle P \rangle ::= \langle P \rangle + \langle symbol \rangle$$

$$\langle P \rangle ::= \langle symbol \rangle + \langle P \rangle$$

5.4 Implementacja AST

Zaimplementowany AST składa się ze struktury `ast_node`. Jest to główny typ węzła, zawierający niektóre zbędne informacje dla każdego typu węzła AST, i przechowujący konkretny węzeł jako wskaźnik.

```
struct ast_node {
    enum ast_type  type;      /* Rozrozniamy typ według tej flagi */
    void          *ast;      /* ast_num, ast_for, ast_while, et cetera */
    uint16_t      line_no;
    uint16_t      col_no;
};
```

Konkretne węzły definiujemy w następujący sposób:

```
struct ast_num {
    int32_t value;
};
```

Taki AST stanowi strukturę drzewiastą, mającą wszystkie zalety i wady drzew jako struktur danych. Mając takie drzewo, jesteśmy w stanie prowadzić zwykłe przeszukiwanie w głąb i wszerz. W danym przypadku taki algorytm się nazywa **AST visitor**. Dokładnie w ten sposób działa każda z przedstawionych niżej analiz semantycznych oraz generacja kodu pośredniego.

Algorithm 1 Przeszukiwanie AST

```
1: procedure DFS(AST)
2:   for each child node Child of AST do
3:     DFS(Child)
4:   end for
5: end procedure
```

5.5 Implementacja analizatora składniowego

W danym przypadku, analizator składniowy jest napisany ręcznie, chociaż są narzędzia od projektu GNU, takie jak GNU Bison i UNIX'owe, takie jak YACC. Niniejszy analizator jest napisany bez pomocy tych programów, aby jawnie pokazać, jak się przekładają produkcje BNF na język C.

Aby poradzić sobie z zadaniem pisania takiego analizatora, możemy zauważyć, że zadanie to sprowadza się do implementacji każdej produkcji gramatycznej osobno.

5.6 Reprezentacja wizualna AST

Jest pokazana też implementacja **visitor**'u, pozwalającego na przeprowadzenie AST do formy tekstowej. Do tego służy funkcja `ast_dump()`. Przyjmuje ona wskaźnik do węzła drzewa i działa według algorytmu DFS, opisanego wyżej, przy tym pisząc tekstową formę węzłów do pliku (ewentualnie, do `stdout`). Funkcjonalność ta jest bardzo ważna do prowadzenia testów jednostkowych samego AST oraz analizatora skła-

dniowego. Niżej pokazany jest przykładowy wynik działania tej funkcji.

```
CompoundStmt <line:0, col:0>  
  StructDecl <line:9, col:1> 'custom'  
    CompoundStmt <line:9, col:1>  
      VarDecl <line:10, col:5> int 'a'  
      VarDecl <line:11, col:5> int 'b'  
      VarDecl <line:12, col:5> int 'c'  
      ArrayDecl <line:13, col:5> char [1000] 'mem'  
      VarDecl <line:14, col:5> struct string 'description'
```

6 Analiza semantyczna

Aby zapewnić poprawność napisanego kodu, stosuje się wiele rodzajów analiz. Niniejszy kompilator dysponuje trzema:

- Analiza nieużytych zmiennych, oraz zmiennych, które są zdefiniowane, ale nie zostały użyte
- Analiza poprawności typów
- Analiza prawidłowego użycia funkcji

6.1 Analiza nieużywanych zmiennych

Podamy przykłady kodu prowadzącego do odpowiednich ostrzeżeń

```
void f() {
    int argument = 0; // Warning: unused variable 'argument'
}

void f(int argument) { // Warning: unused variable 'argument'
}

void f() {
    int argument = 0;
    ++argument; // Warning: variable 'argument' written, but never read
}
```

Rzecz polega na przejściu drzewa syntaktycznego i zwiększania liczników **read_uses** i **write_uses** dla każdego węzła typu **ast_sym**.

Algorytm operuje na blokach kodu, zawartego w { ... }. Po przejściu każdego bloku (w tym rekurencyjnie), analiza jest wykonana w następujący sposób:

Algorithm 2 Wyszukiwanie nieużywanych zmiennych

```
1: procedure ANALYZE(AST)
2:   Set ← all declarations at current scope depth
3:   for each collected declaration Use in Set do
4:     if Use is not a function & Use.ReadUses is 0 then
5:       Emit warning
6:     end if
7:   end for
8: end procedure
```

Do analizy nieużywanych funkcji stosuje się ten sam algorytm. Jedyne, co jest wtedy zmienione – sprawdzenie, czy nazwa rozpatrywanej funkcji nie jest **main**. Funkcja **main** jest wywołana automatycznie.

7 System typów

7.1 Opis

Wiele zasad, dotyczących pracy z typami mogą być precyzyjnie opisane zasadami typów (**Typing rules**). Jest to notacja matematyczna, znaczenie której niżej wyjaśnimy.

Kluczowym pojęciem w tej notacji jest **statyczne środowisko typów** (**static typing environment**). Oznacza się ono symbolem Γ . Mówimy, że to środowisko jest skonstruowane poprawnie pisząc

$$\Gamma \vdash \diamond$$

Mówimy, że zmienna V ma typ T w środowisku Γ pisząc

$$\Gamma \vdash V : T$$

Kreska pozioma mówi o tym, że zdanie wyżej jest konieczne, aby zaszło zdanie niżej

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash V : T}$$

Zauważmy, że notacja ta jest mocnym narzędziem, pozwalającym opisać dość złożone systemy typów dla takich języków jak **C++** i **Haskell**.

7.2 Definicja systemu

Opiszmy teraz system typów w naszym języku

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \text{true} : \text{bool}}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \text{false} : \text{bool}}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash n : \text{int}}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash c : \text{char}}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash x : \text{float}}$$

Oznaczmy dla $\mathbb{N}, \mathbb{R} : \oplus \in \{=, +, -, *, /, <, >, \leq, \geq, ==, \neq, ||, \&\&\}$, wtedy

$$\frac{\Gamma \vdash e_l : float \quad \Gamma \vdash e_r : float}{\Gamma \vdash e_l \oplus e_r : float}$$

Dodamy do \oplus operacje tylko dla $\mathbb{N} : \oplus \cup \{ |, \&, ^, <<, >>, \% \}$, wtedy

$$\frac{\Gamma \vdash e_l : int \quad \Gamma \vdash e_r : int}{\Gamma \vdash e_l \oplus e_r : int}$$

$$\frac{\Gamma \vdash e_l : int \quad \Gamma \vdash e_r : char}{\Gamma \vdash e_l \oplus e_r : char}$$

Wprowadźmy reguły niejawnej konwersji, które są niezbędne przy sprawdzaniu w warunku logicznym wyniku operacji arytmetycznej, zwracającej typ różny od `bool`. Oznaczmy reguły dla typów `int`, `char` i `float`.

$$\frac{\Gamma \vdash e : int}{\Gamma \vdash e : bool}$$

$$\frac{\Gamma \vdash e : char}{\Gamma \vdash e : bool}$$

$$\frac{\Gamma \vdash e : float}{\Gamma \vdash e : bool}$$

Wprowadźmy także reguły do operacji wskaźnikowych. Oznaczmy $\oplus \in \{=, +, -, \leq, \geq, ==, \neq\}$, wtedy

$$\frac{\Gamma \vdash e_l : int * \quad \Gamma \vdash e_r : int *}{\Gamma \vdash e_l \oplus e_r : int *}$$

$$\frac{\Gamma \vdash e_l : char * \quad \Gamma \vdash e_r : char *}{\Gamma \vdash e_l \oplus e_r : char *}$$

$$\frac{\Gamma \vdash e_l : float * \quad \Gamma \vdash e_r : float *}{\Gamma \vdash e_l \oplus e_r : float *}$$

$$\frac{\Gamma \vdash e_l : bool * \quad \Gamma \vdash e_r : bool *}{\Gamma \vdash e_l \oplus e_r : bool *}$$

Mając taką konwersję, możemy wprowadzić reguły do konstrukcji warunkowych:

$$\frac{\Gamma \vdash condition : bool \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if (condition) } \{ e_1 \} \text{ else } \{ e_2 \} : \tau}$$

$$\frac{\Gamma \vdash condition : bool \quad \Gamma \vdash e : \tau}{\Gamma \vdash \text{while (condition) } \{ e \} : \tau}$$

$$\frac{\Gamma \vdash condition : bool \quad \Gamma \vdash e : \tau}{\Gamma \vdash \text{do } \{ e \} \text{ while (condition) : } \tau}$$

$$\frac{\Gamma \vdash init : \tau_1 \quad \Gamma \vdash condition : bool \quad \Gamma \vdash increment : \tau_2 \quad \Gamma \vdash e : \tau_3}{\Gamma \vdash \text{for (init; condition; increment) } \{ e \} : \tau_3}$$

8 Generacja kodu pośredniego

Kod pośredni – jest to język, składający się z elementarnych operacji nad danymi, takimi jak arytmetyczne operacje, zapisanie do komórki pamięci.

Im prostszy ten język jest, tym prostsze są algorytmy do analizy, optymalizacji i generacji dalszych warstw pośrednich.

Istnieje wiele różnych podobnych do assemblera języków (Intermediate representation, **IR**), służącego do generacji kodu maszynowego (LLVM IR, GIMPLE, FIRM). Jednak, niniejszy język implementuje własny IR z kilku powodów:

- Jest prostszy
- Ma prostszy interfejs programistyczny
- Nie stanowi dodatkowych zależności jako biblioteki
- Ma na celu pokazanie metod na tworzenie takiego języka i operacje nad nimi

9 Optymalizacje kodu pośredniego

9.1 Constant propagation

Został zaimplementowany algorytm przeliczania stałych. Rozpatruje on każdy block CFG osobno, co pozwala na przeprowadzenie lokalnych optymalizacji. Niniejszy algorytm nie usuwa zmiennych, których nie potrzebuje po obliczeniach dlatego, że to jest obowiązek innej optymalizacji (**dead code elimination**).

Algorithm 3 Przeliczenie zmiennych stałych

```

1: procedure CONSTPROP(AST)
2:   Consts  $\leftarrow \emptyset$ 
3:   for each statement S across all CFG blocks do
4:     if new CFG block is reached then
5:       Consts  $\leftarrow \emptyset$  ▷ Reset optimizer state
6:     end if
7:     if S is binary then
8:       Consts  $\leftarrow$  Consts  $\cup$  computed value of binary expression
9:     end if
10:    if S is store then
11:      if S stores immediate then
12:        if Consts has value for S.Sym then
13:          Update Consts with new value
14:        else
15:          Consts  $\leftarrow$  Consts  $\cup$  value
16:        end if
17:      end if
18:      if S stores symbol then
19:        if Consts has value for S.Sym then
20:          Update Consts with new value
21:        else
22:          Consts  $\leftarrow$  Consts  $\setminus \{ \text{value} \}$ 
23:        end if
24:      end if
25:    end if
26:  end for
27: end procedure

```

10 Interpreter

11 Annex: Gramatyka w BNF

$\langle \text{program} \rangle$	$::= (\langle \text{function-decl} \rangle \mid \langle \text{structure-decl} \rangle)^*$
$\langle \text{structure-decl} \rangle$	$::= \text{struct } \{ \langle \text{structure-decl-list} \rangle \}$
$\langle \text{structure-decl-list} \rangle$	$::= (\langle \text{decl-without-initialiser} \rangle ;$ $\mid \langle \text{structure-decl} \rangle ;)^*$
$\langle \text{function-decl} \rangle$	$::= \langle \text{ret-type} \rangle \langle \text{id} \rangle (\langle \text{parameter-list-opt} \rangle) \{ \langle \text{stmt} \rangle^* \}$
$\langle \text{ret-type} \rangle$	$::= \langle \text{type} \rangle$ $\mid \langle \text{void-type} \rangle$
$\langle \text{type} \rangle$	$::= \text{int}$ $\mid \text{float}$ $\mid \text{char}$ $\mid \text{string}$ $\mid \text{boolean}$
$\langle \text{void-type} \rangle$	$::= \text{void}$
$\langle \text{constant} \rangle$	$::= \langle \text{integral-literal} \rangle$ $\mid \langle \text{floating-literal} \rangle$ $\mid \langle \text{string-literal} \rangle$ $\mid \langle \text{char-literal} \rangle$ $\mid \langle \text{boolean-literal} \rangle$
$\langle \text{integral-literal} \rangle$	$::= \langle \text{digit} \rangle^*$
$\langle \text{floating-literal} \rangle$	$::= \langle \text{digit} \rangle^* . \langle \text{digit} \rangle^*$
$\langle \text{string-literal} \rangle$	$::= \text{' ' (x00000000-x0010FFFF) * ' '}$
$\langle \text{char-literal} \rangle$	$::= \text{'ASCII(0)-ASCII(127)'}$
$\langle \text{boolean-literal} \rangle$	$::= \text{true}$ $\mid \text{false}$
$\langle \text{alpha} \rangle$	$::= \text{a} \mid \text{b} \mid \dots \mid \text{z} \mid _$
$\langle \text{digit} \rangle$	$::= 0 \mid 1 \mid \dots \mid 9$
$\langle \text{id} \rangle$	$::= \langle \text{alpha} \rangle (\langle \text{alpha} \rangle \mid \langle \text{digit} \rangle)^*$
$\langle \text{array-decl} \rangle$	$::= \langle \text{type} \rangle (*)^* \langle \text{id} \rangle [\langle \text{integral-literal} \rangle]$
$\langle \text{var-decl} \rangle$	$::= \langle \text{type} \rangle (*)^* \langle \text{id} \rangle = \langle \text{logical-or-stmt} \rangle ;$
$\langle \text{structure-var-decl} \rangle$	$::= \langle \text{id} \rangle (*)^* \langle \text{id} \rangle$

$\langle decl \rangle$	$::= \langle var-decl \rangle$ $\langle array-decl \rangle$ $\langle structure-var-decl \rangle$
$\langle decl-without-initialiser \rangle$	$::= \langle type \rangle (*) * \langle id \rangle$ $\langle array-decl \rangle$ $\langle structure-var-decl \rangle$
$\langle parameter-list \rangle$	$::= \langle decl-without-initialiser \rangle , \langle parameter-list \rangle$ $\langle decl-without-initialiser \rangle$
$\langle parameter-list-opt \rangle$	$::= \langle parameter-list \rangle \mid \epsilon$
$\langle stmt \rangle$	$::= \langle block-stmt \rangle$ $\langle selection-stmt \rangle$ $\langle iteration-stmt \rangle$ $\langle jump-stmt \rangle$ $\langle decl \rangle$ $\langle expr \rangle$ $\langle assignment-stmt \rangle$ $\langle primary-stmt \rangle$
$\langle member-access-stmt \rangle$	$::= \langle id \rangle . \langle member-access-stmt \rangle$ $\langle id \rangle . \langle id \rangle$
$\langle iteration-stmt \rangle$	$::= \langle stmt \rangle$ break ; continue ;
$\langle block-stmt \rangle$	$::= \{ \langle stmt \rangle * \}$
$\langle iteration-block-stmt \rangle$	$::= \{ \langle iteration-stmt \rangle * \}$
$\langle selection-stmt \rangle$	$::= \text{if } (\langle expr \rangle) \langle block-stmt \rangle$ $\text{if } (\langle expr \rangle) \langle block-stmt \rangle \text{ else } \langle block-stmt \rangle$
$\langle iteration-stmt \rangle$	$::= \text{for } (\langle expr-opt \rangle ; \langle expr-opt \rangle ; \langle expr-opt \rangle) \langle iteration-block-stmt \rangle$ $\text{for } (\langle decl \rangle : \langle symbol-stmt \rangle) \langle iteration-block-stmt \rangle$ $\text{while } (\langle expr \rangle) \langle iteration-block-stmt \rangle$ $\text{do } \langle iteration-block-stmt \rangle \text{ while } (\langle expr \rangle) ;$
$\langle jump-stmt \rangle$	$::= \text{return } \langle expr \rangle ? ;$
$\langle assignment-op \rangle$	$::= =$ $==$ $/=$ $\% =$ $+=$ $-=$ $<<=$

	>>=
	&=
	=
	^=
$\langle expr \rangle$	$::= \langle assignment-stmt \rangle$ $\langle var-decl \rangle$
$\langle expr-opt \rangle$	$::= \langle expr \rangle \mid \epsilon$
$\langle assignment-stmt \rangle$	$::= \langle logical-or-stmt \rangle$ $\langle logical-or-stmt \rangle \langle assignment-op \rangle \langle assignment-stmt \rangle$
$\langle logical-or-stmt \rangle$	$::= \langle logical-and-stmt \rangle$ $\langle logical-and-stmt \rangle \parallel \langle logical-or-stmt \rangle$
$\langle logical-and-stmt \rangle$	$::= \langle inclusive-or-stmt \rangle$ $\langle inclusive-or-stmt \rangle \&\& \langle logical-and-stmt \rangle$
$\langle inclusive-or-stmt \rangle$	$::= \langle exclusive-or-stmt \rangle$ $\langle exclusive-or-stmt \rangle \mid \langle inclusive-or-stmt \rangle$
$\langle exclusive-or-stmt \rangle$	$::= \langle and-stmt \rangle$ $\langle and-stmt \rangle \wedge \langle exclusive-or-stmt \rangle$
$\langle and-stmt \rangle$	$::= \langle equality-stmt \rangle$ $\langle equality-stmt \rangle \& \langle and-stmt \rangle$
$\langle equality-stmt \rangle$	$::= \langle relational-stmt \rangle$ $\langle relational-stmt \rangle == \langle equality-stmt \rangle$ $\langle relational-stmt \rangle != \langle equality-stmt \rangle$
$\langle relational-stmt \rangle$	$::= \langle shift-stmt \rangle$ $\langle shift-stmt \rangle > \langle relational-stmt \rangle$ $\langle shift-stmt \rangle < \langle relational-stmt \rangle$ $\langle shift-stmt \rangle >= \langle relational-stmt \rangle$ $\langle shift-stmt \rangle <= \langle relational-stmt \rangle$
$\langle shift-stmt \rangle$	$::= \langle additive-stmt \rangle$ $\langle additive-stmt \rangle << \langle shift-stmt \rangle$ $\langle additive-stmt \rangle >> \langle shift-stmt \rangle$
$\langle additive-stmt \rangle$	$::= \langle multiplicative-stmt \rangle$ $\langle multiplicative-stmt \rangle + \langle additive-stmt \rangle$ $\langle multiplicative-stmt \rangle - \langle additive-stmt \rangle$
$\langle multiplicative-stmt \rangle$	$::= \langle prefix-unary-stmt \rangle$ $\langle prefix-unary-stmt \rangle * \langle multiplicative-stmt \rangle$ $\langle prefix-unary-stmt \rangle / \langle multiplicative-stmt \rangle$ $\langle prefix-unary-stmt \rangle \% \langle multiplicative-stmt \rangle$

$\langle \text{prefix-unary-stmt} \rangle$	$::=$	$\langle \text{postfix-unary-stmt} \rangle$ $++ \langle \text{postfix-unary-stmt} \rangle$ $-- \langle \text{postfix-unary-stmt} \rangle$ $* \langle \text{postfix-unary-stmt} \rangle$ $\& \langle \text{postfix-unary-stmt} \rangle$ $! \langle \text{postfix-unary-stmt} \rangle$
$\langle \text{postfix-unary-stmt} \rangle$	$::=$	$\langle \text{primary-stmt} \rangle$ $\langle \text{primary-stmt} \rangle ++$ $\langle \text{primary-stmt} \rangle --$
$\langle \text{primary-stmt} \rangle$	$::=$	$\langle \text{constant} \rangle$ $\langle \text{symbol-stmt} \rangle$ $(\langle \text{logical-or-stmt} \rangle)$
$\langle \text{symbol-stmt} \rangle$	$::=$	$\langle \text{function-call-stmt} \rangle$ $\langle \text{array-access-stmt} \rangle$ $\langle \text{member-access-stmt} \rangle$ $\langle \text{id} \rangle$
$\langle \text{array-access-stmt} \rangle$	$::=$	$\langle \text{id} \rangle ([\langle \text{expr} \rangle])^*$
$\langle \text{function-call-arg-list} \rangle$	$::=$	$\langle \text{logical-or-stmt} \rangle , \langle \text{function-call-arg-list} \rangle$ $\langle \text{logical-or-stmt} \rangle$
$\langle \text{function-call-arg-list-opt} \rangle$	$::=$	$\langle \text{function-call-arg-list} \rangle \mid \epsilon$
$\langle \text{function-call-expr} \rangle$	$::=$	$\langle \text{id} \rangle (\langle \text{function-call-arg-list-opt} \rangle)$

Literatura

- [1] <https://www.bates.edu/biology/files/2010/06/How-to-Write-Guide-v10-2014.pdf>
- [2] Bauer, Friedrich Ludwig, *Compiler construction*, 1974, Berlin, ISBN: 3-540-06958-5
- [3] <http://lucacardelli.name/papers/typesystems.pdf>
- [4] <https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html>
- [5] <https://llvm.org/docs/LangRef.html>
- [6] <https://github.com/libfirm/libfirm>