Kompilator języka strukturalnego

David Korenchuk

August, 23, 2023

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 wieku został stworzony system szyfrowania informacji na podstawie zliczania ilości liter w tekscie.
- W **XVII** wieku powstał suwak logarytmiczny, potrzebny do ułatwienia działań matematycznych.
- W tym samym **XVII** wieku powstał jeden z pierszych kalkulatorów mechanicznych **Pascalina**. Jest to narzędzie do wykonania operacji arytmetycznych na podstawie ruchu kół 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 rozwiązania 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 opisaniu 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

```
0 void f() {
1   return argument + 1;
2 }
```

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.

```
0 void f() {
1   data *d;
2 }
```

4 Analiza leksykalna

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

4.1 Wyrażenia regularne

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

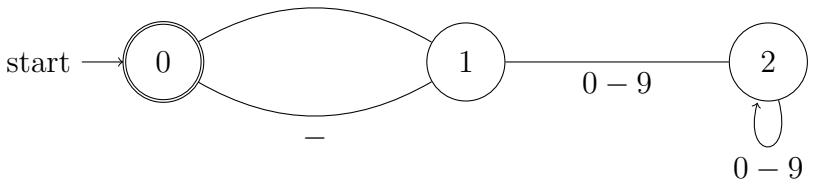
Mówimy, że wyrażenie regularne określone nad alfabetem Σ, jeżeli zachodzą następujące warunki:

- $\emptyset$ – 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 konkatenację 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

```
0 %{
1 #include "portrzebny-do-analizy-plik.h"
2
3 /* Kod w jezyku C. */
4 %}
5
6 /* Opcje flex */
7 %option noyywrap nounput noinput
8 %option yylineno
9
10 %% /* Reguly w postaci wyrazen regularnych. */
11
12 /*************************************************************/
13 /* Wzorzec                    | Akcja przy znalezieniu takiego wzorcu */
14 /*************************************************************/
15 -?[0-9]+                        LEX_CONSUME_WORD(TOK_INTEGRAL_LITERAL)
16 -?[0-9]+\.[0-9]+                LEX_CONSUME_WORD(TOK_FLOATING_POINT_LITERAL)
17 \"([^\"\\]*(\\.[^\"\\]*)*)\"    LEX_CONSUME_WORD(TOK_STRING_LITERAL)
18 \'.\'                          LEX_CONSUME_WORD(TOK_CHAR_LITERAL)
19
20 .                              { /* Znaleziony niewiadomy znak.
21                                      Zglosic blad.
22                                   */ }
23 %%
```

Zauważmy, że flex próbuje szukać wzorców w tekscie dokładnie w takiej kolejności, która jest podana w jego kodzie. Dlatego często robią ostatnią regułe 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:

⟨*program*⟩ ::= (⟨*function-decl*⟩ | ⟨*structure-decl*⟩)*

⟨*var-decl*⟩ ::= ⟨*type*⟩ (`*`)* ⟨*id*⟩ `=` ⟨*logical-or-stmt*⟩ `;`

⟨*stmt*⟩ ::= ⟨*block-stmt*⟩
| ⟨*selection-stmt*⟩
| ⟨*iteration-stmt*⟩
| ⟨*jump-stmt*⟩
| ⟨*decl*⟩
| ⟨*expr*⟩
| ⟨*assignment-stmt*⟩
| ⟨*primary-stmt*⟩

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

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 tą tablicy, 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.

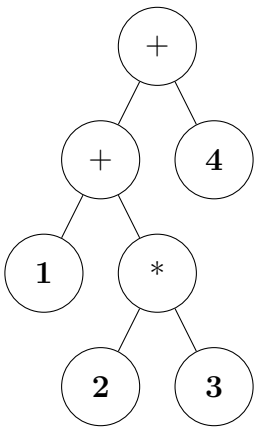
Podamy gramatykę dla przykładu powyżej (`1 + 2 * 3 + 4`). Musimy wprowadzić dwa poziomy priorytetów, aby prawidłowo zachować kolejność operacji mnożenia i dodawania.

⟨*additive-stmt*⟩ ::= ⟨*multiplicative-stmt*⟩
| ⟨*multiplicative-stmt*⟩ `+` ⟨*additive-stmt*⟩
| ⟨*multiplicative-stmt*⟩ `-` ⟨*additive-stmt*⟩

⟨*multiplicative-stmt*⟩ ::= ⟨*prefix-unary-stmt*⟩
| ⟨*prefix-unary-stmt*⟩ * ⟨*multiplicative-stmt*⟩
| ⟨*prefix-unary-stmt*⟩ / ⟨*multiplicative-stmt*⟩
| ⟨*prefix-unary-stmt*⟩ % ⟨*multiplicative-stmt*⟩

	First	**Follow**
<additive-stmt>	0-9	+, -
<multiplicative-stmt>	0-9	*, /
<prefix-unary-stmt>	0-9	ϵ

	0-9	+	-	*	/	$
<additive-stmt>		<mul> + <add>	<mul> - <add>			<mul>
<multiplicative-stmt>				<una> * <mul>	<una> / <mul>	<una>
<prefix-unary-stmt>	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 tą samą produkcję, co prowadzi do rekurencji nieskończonej.

Rekuręcja lewa może wyglądać następująco:

⟨*factor*⟩ ::= ⟨*factor*⟩ '+' ⟨*term*⟩

Kod, wykonujący tą reguły będzie miał postać:

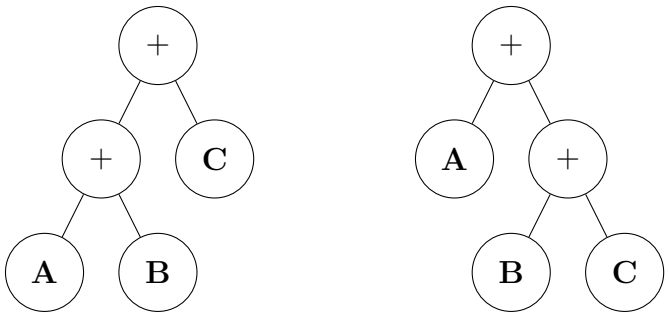
```
0 void factor() {
1   factor(); // Rekurencja bez zadnego warunku wyjscia
2   consume('+');
3   term();
4 }
```

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

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

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**.

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

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

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.

```
0 struct ast_node {
1     enum ast_type  type;     /* Rozrozniamy typ wedlug tej flagi */
2     void          *ast;      /* ast_num, ast_for, ast_while, et cetera */
3     uint16_t       line_no;
4     uint16_t       col_no;
5 };
```

Konkretne węzły definiujemy w następujący sposób:

```
0 struct ast_num {
1     int32_t value;
2 };
```

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 drewa 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 analizatoru składniowego. Niżej pokazany jest przykładowy wynik działania tej funkcji.

```
0 CompoundStmt <line:0, col:0>
1   StructDecl <line:9, col:1> ‘custom‘
2     CompoundStmt <line:9, col:1>
3       VarDecl <line:10, col:5> int ‘a‘
4       VarDecl <line:11, col:5> int ‘b‘
5       VarDecl <line:12, col:5> int ‘c‘
6       ArrayDecl <line:13, col:5> char [1000] ‘mem‘
7       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

```
0 void f() {
1   int argument = 0; // Warning: unused variable ‘argument‘
2 }
```

```
0 void f(int argument) { // Warning: unused variable ‘argument‘
1
2 }
```

```
0 void f() {
1   int argument = 0;
2   ++argument; // Warning: variable ‘argument‘ written, but never read
3 }
```

Rzecz polega na przejściu drzewa syntaksycznego 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 tem 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, stworzona przez **Per Martin-Löf**'a, szwedzkiego matematyka.

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

$$\Gamma \vdash \mathrm{V} : T$$

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

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

Zauważmy, że notacja ta jest mocnyn 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

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathrm{true} : \mathit{bool}} \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathrm{false} : \mathit{bool}}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathrm{n} : \mathit{int}} \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathrm{c} : \mathit{char}} \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathrm{x} : \mathit{float}}$$

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

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

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

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

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`.

$$\frac{\Gamma \vdash \mathrm{e} : \mathit{int}}{\Gamma \vdash \mathrm{e} : \mathit{bool}} \qquad \frac{\Gamma \vdash \mathrm{e} : \mathit{char}}{\Gamma \vdash \mathrm{e} : \mathit{bool}} \qquad \frac{\Gamma \vdash \mathrm{e} : \mathit{float}}{\Gamma \vdash \mathrm{e} : \mathit{bool}}$$

Wprowadźmy także reguły do operacji wskaźnikowych. Oznaczmy $\oplus \in \{+, -\}$ i τ jako wskaźnik dowolnego typu, wtedy

$$\frac{\Gamma \vdash e_l : \tau * \qquad \Gamma \vdash e_r : \tau *}{\Gamma \vdash e_l \oplus e_r : \tau *} \qquad \frac{\Gamma \vdash e_l : \tau * \qquad \Gamma \vdash e_r : \mathit{int}}{\Gamma \vdash e_l \oplus e_r : \tau *} \qquad \frac{\Gamma \vdash e_l : \mathit{int} \qquad \Gamma \vdash e_r : \tau *}{\Gamma \vdash e_l \oplus e_r : \tau *}$$

Dla $\oplus \in \{==, ! =, <, >, \leqslant, \geqslant\}$

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

Mając taką konwersję, możemy wprowadzić reguły do konstrukcji warunkowych:

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

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

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

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

8.1 Algorytm

Generator IR przyjmuje AST jako wejście. Stworzenie instrukcji pośrednich polega na rekurencyjnym przejściu tego drzewa oraz tłumaczeniu danych o programie z poziomu drzewa (w zasadzie elementów syntaksycznych) na język niskiego poziomu. W zależności od węzła musi być stworzony kod, zachowujący się zgodnie z oczekiwaniem użytkownika.

Naprzykład, pętla while (po lewej stronie) zostanie przetłumaczona na kod niskiego poziomu (po prawej stronie).

```
0  int main() {
1      int i = 1;
2      while (i < 10 && i != 0)
3      {
4          int j = i;
5          ++j;
6          ++i;
7      }
8      return 0;
9  }
10 .
11 .
12 .
13 .
14 .
15 .
16 .
```

```
0  fun main():
1          0:   int t0
2          1:   t0 = 1
3          2:   | int t1
4          3:   | int t2
5          4:   | t2 = t0 < 10
6          5:   | int t3
7          6:   | t3 = t0 != 0
8          7:   | t1 = t2 && t3
9          8:   | if t1 != 0 goto L10
10         9:   | jmp L15
11        10:   | int t4
12        11:   | t4 = t0
13        12:   | t4 = t4 + 1
14        13:   | t0 = t0 + 1
15        14:   | jmp L2
16        15:   ret 0
```

Widać, że złożona semantycznie konstrukcja `while` została przedstawiona za pomocą prostszych do wykonania operacji: stworzyć zmienną o rozmiarze N bajtów, zapisać do niej wynik operacji binarnej, wykonać skok do instrukcji o pewnym indeksie. Analogicznie przedstawione są inne części języka (`for`, `do while`, `if`). Oczywiście, wszystkie konstrukcje mogą być zagnieżdżone dowolną ilośc razy, przy tym zachowawszy liniową strukturę IR.

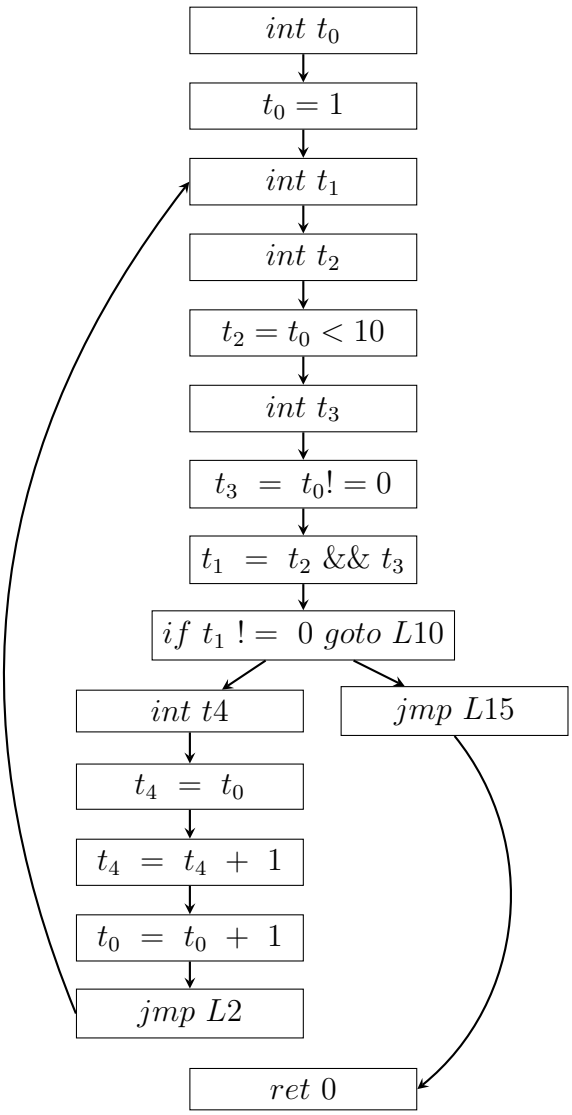
8.2 Instrukcje

Zdefiniowane w implementacji struktury dzielą się na instrukcje i typy danych.

Instrukcja	**Opis**
ir_alloca	Alokacja określonej ilości bajtów na stosie.
ir_alloca_array	Alokacja określonej ilości bajtów na stosie, pomnożonej przez rozmiar tablicy.
ir_store	Zapisanie wartości do zmiennej lub do tablicy o wskazanym indeksie.
ir_jump	Przekazanie przepływu sterowania do instrukcji o wskazanym indeksie.
ir_cond	Instrukcja warunkowa. Wykonuje skok (taki sam, jak `ir_jump`), jeżeli warunek jest spełniony. Inaczej sterowanie się przekazuje do następnej instrukcji.
ir_ret	Wyjście z funkcji. Może mieć lub nie mieć wartość zwracaną.
ir_fn_call	Wywołanie funkcji.

Typ	**Opis**
ir_imm	Wartość (int, char, bool).
ir_string	Łańcuch znaków.
ir_sym	Indeks zmiennej.
ir_bin	Operacja binarna (dwuargumentowa).
ir_member	Operacja dostępu do indeksu tablicy.
ir_type_decl	Definicja typu.
ir_fn_decl	Definicja funkcji.

8.3 Generacja grafu sterowania



Wygenerowana poprzednio warstwa poprzednia zawiera wszystkie informacje, dotyczące operacji nad danymi, ale brakuje jeszcze informacji o przepływie sterowania. Aby wiedzieć, która instrukcja się wykonuje po której, musimy stworzyć związek między poprzednią i następną instrukcją, który określa się następująco

- Instrukcja warunkowa ma dwóch następników. Jeden jest wykonany w przypadku spełnionengo warunku, a drugi – gdy warunek nie zaszedł.
- Instrukcja skokowa ma jednego następnika, niekoniecznie będącego następnikiem na liście kodu. Jeżeli index instrukcji docelowej jest < od instrukcji skoku, powstaje **krawędź powrotna** (**back edge**). Wszystkie inne krawędzi są **skierowane w przód** (**forward edge**).

W przykładzie, pokazanym po lewej stronie, jest jedna krawędź powrotna, wiodąca od *jmp L2* do *int* t_1.

Otrzymany graf jest użyteczny przy wielu rodzajach optymalizacji. Naprzykład, przy usuwaniu kodu nieosiądalnego oraz analizie zależności danych.

9 Optymalizacje (teoria)

Optymalizacja – to zmiana kodu programu, mająca na celu polepszyć wydajność albo inne cechy programu. Najważniejym z kriteria optymalizacji jest utrzymanie całej struktury działania programu, takiej,

jak chce programista. Nie wolno przeprowadzać wiodące do niespodziewanych lub niepoprawnych wyników optymalizacje.

Istnieje wiele rodzajów optymalizacji, gdzie każda wymaga więcej lub mniej założeń i matematyki. Jeżeli chodzi o matematykę, to główną rolę w optymalizacji pełni **teoria grafów**. Jednym z pierwszych naukowców, kto zadecydował wprowadzić modele grafowe do kompilatorów był **Robert Tarjan**. Wprowadził także on algorytm do obliczenia drzewa dominatorów (dominator tree).

9.1 Definicje

Każdy program posiadą taką cechę jak **przepływ sterowania**, i może ona być precyzyjnie wyrażona **grafem przepływu sterowania** (Control Flow Graph). Program rozpatrywany jest jako graf skierowany, posiadający wierzchołek startowy, będący pierwszą instrukcją w programie. Krawędzie reprezentują przejścia pomiędzy instrukcjami. Zauważmy, że każda instrukcja może mieć wiele krawędzi wejściowych i wyjściowych, tworząc **gałęzi** w wykonaniu programu.

Oznaczmy kilka ważnych pojęć.

Niech $G = (V, E, s)$ – graf skierowany. V – zbiór wierzchołków. E – zbiór krawędzi. s – węzęł początkowy. $G' = (V', E') \subseteq G$ – podgraf, gdzie każdy wierzchołek $v \in G'$ jest nieosiągalny z dowolnej ścieżki $(s, ..., v) \in G$. Zauważmy, że G' może być grafem rozłącznym.

$$V' = \{\ v \mid \forall\ v \nexists\ (s, ..., v)\ \}$$

Niech $G = (V, E, s)$ – graf skierowany, zdefiniowany powyżej. Graf zależności danych – graf $G' = (V', E', D')$. Wtedy $D' = \{\ d, d' \in V' \mid d \to d'\ \}$. Przez $d \to d'$ oznaczona zależnośc d od wyniku obliczenia d'. Zbiór D' reprezentuje takie zależności, przy czym dodatkowo musi być spełnione

$$d, d' \in V',\ v \to v' \iff \exists\ (v', ..., v) \in G'$$

9.2 SSA

Static Assignment Form (SSA) – ważna forma reprezentacji programu. Nazywa się tak program, w którym każda zmienna jest zapisana dokładnie jeden raz. Odkrywa to wiele możliwości do prowadzenia optymalizacji, na przykład jest to propagacja zmiennych stałych i alokacja rejestrów.
Konstrukcja SSA składa się z kilku etapów:

- Obliczenie drzewa dominatorów (dominator tree).
- Obliczenie granicy dominacji (dominance frontier).
- Wstawianie ϕ-funkcji (zob. niżej).

Istnieje przynajmniej dwa algorytmy do obliczenia SSA na różne sposoby. Pierwszy autorstwa **Ron Cytron** i kilku innych matematyków (An Efficient Method of Computing Static Single Assignment Form). Drugi autorstwa **Quan Hoang Nguyen** (Computing SSA Form with Matrices). Został zaimplementowany pierwszy algorytm, skoro jest prostrzy i wymaga mniej (chociaż też dużo) kultury matematycznej. Nie będziemy udowadniać tutaj całego algorytmu, pokażemy tylko główne jego idee.

9.3 SSA: drzewo dominatorów

Do obliczenia drzewa dominatorów używa się algorytm, stworzony przez dwóch matematyków – algorytm **Lengauer-Tarjan**'u. Jest on profesjonalnie zbudowany i udowodniony matematycznie. Składa się on z trzech części

1. Przejście grafu w głąb (zwykły DFS).
2. Przejście wyniku DFS w odwrotnej kolejności, zatem przeliczanie półdominatorów (**semidominators**).
3. Za pomocą półdominatorów definicja dominatorów.

9.4 SSA: granica dominacji

9.5 SSA: ϕ-funkcje

10 Optymalizacje (implementacja)

10.1 Unreachable code elimination

Usuwanie kodu nieosiągalnego polega na dwóch krokach.

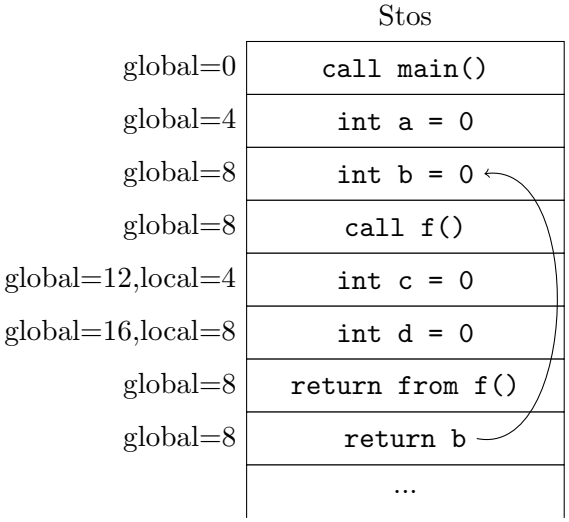
- Obejście grafu sterowania programu (**CFG**)
- Usuwanie wszystkich instrukcji, do których nie ma żadnych wejściowych krawędzi.

Algorytm polega na zlalezieniu takich **podgrafów** grafu sterowania programem, do których nie prowadzi żadna z krawędzi.

Algorithm 3 Usuwanie kodu nieosiągalnego

```
1: procedure ELIMINATE(CFG)
2:     Visited ← ∅
3:     TRAVERSE(Visited, CFG, First(CFG))
4:     Unvisited ← CFG \ Visited
5:     CUT(Unvisited, CFG)
6: end procedure
7:
8: procedure TRAVERSE(Visited, CFG, IR)
9:     Visited[IR] ← 1
10:    for each control flow successor of IR do
11:        TRAVERSE(Visited, CFG, Succ(IR))
12:    end for
13: end procedure
14:
15: procedure CUT(Unvisited, CFG)
16:    for each unvisited statement do
17:        Remove statement from IR
18:    end for
19: end procedure
```

11 Interpreter



Interpreter – program, który produkuje dane wyjściowe zgodnie z zasadami semantycznymi, które są opisane językiem. W naszym przypadku, interpreter przetwarza wygenerowany poprzednio IR.

Niniejszy interpreter działa używając stosu. Jest to struktura danych, pozwalająca dodawać i usuwać dane tylko z górnej części stosu. Określamy operacje **push** i **pop**. Stos, używany przez interpreter jednak wspiera przegląd wartości o dowolnej pozycji w stosie.

Aby zarządzać pamięcią podczas wykonania, musimy zdecydować, na jaki sposób zaimplementować **adresację** zmiennych. Jest to operacja, która utożsamia nazwę zmiennej z jej adresem w pamięci. Istineije dwie możliości

Statyczna alokacja pamięci – sposób adresacji, przy którym adres każdej zmiennej określony jednoznacznie. Jest użyteczna do początkowych wersji ALGOL i COBOL, które nie wspierają rekurencyjne wywołania funckji.

Dynamiczna alokacja pamięci – spobób adresacji, przy którym niemożliwe jest obliczenie fizycznego adresu zmienneij, skoro stworzona ona może być w funkcji, wywołanej przez inną funkcję bądź samą siebie. Naprzykład, dana funkcja **f()**, która definiuje zmienną **i**, i załóżmy, że wywoła ona się rekurencyjnie. Wtedy podczas pierwszego wywołania zmienna **i** będzie miała adres **0x00**. Podczas drugiego wywołania **0x04**, dalej **0x08**, **0x0C**, ...

12 Annex: Gramatyka w BNF

⟨*program*⟩ ::= (⟨*function-decl*⟩ | ⟨*structure-decl*⟩)*

⟨*structure-decl*⟩ ::= `struct` { ⟨*structure-decl-list*⟩ }

⟨*structure-decl-list*⟩ ::= `(` ⟨*decl-without-initialiser*⟩ `;`
| ⟨*structure-decl*⟩ `; )*`

⟨*function-decl*⟩ ::= ⟨*ret-type*⟩ ⟨*id*⟩ `(` ⟨*parameter-list-opt*⟩ `)` { ⟨*stmt*⟩* }

⟨*ret-type*⟩ ::= ⟨*type*⟩
| ⟨*void-type*⟩

⟨*type*⟩ ::= `int`
| `float`
| `char`
| `string`
| `boolean`

⟨*void-type*⟩ ::= `void`

⟨*constant*⟩ ::= ⟨*integral-literal*⟩
| ⟨*floating-literal*⟩
| ⟨*string-literal*⟩
| ⟨*char-literal*⟩
| ⟨*boolean-literal*⟩

⟨*integral-literal*⟩ ::= ⟨*digit*⟩*

⟨*floating-literal*⟩ ::= ⟨*digit*⟩* `.` ⟨*digit*⟩*

⟨*string-literal*⟩ ::= `''( x00000000-x0010FFFF )*''`

⟨*char-literal*⟩ ::= `'ASCII(0)-ASCII(127)'`

⟨*boolean-literal*⟩ ::= `true`
| `false`

⟨*alpha*⟩ ::= `a` | `b` | `...` | `z` | `_`

⟨*digit*⟩ ::= `0` | `1` | `...` | `9`

⟨*id*⟩ ::= ⟨*alpha*⟩ `(` ⟨*alpha*⟩ | ⟨*digit*⟩ `)*`

⟨*array-decl*⟩ ::= ⟨*type*⟩ `(` * `)*` ⟨*id*⟩ `[` ⟨*integral-literal*⟩ `]`

⟨*var-decl*⟩ ::= ⟨*type*⟩ `(` * `)*` ⟨*id*⟩ `=` ⟨*logical-or-stmt*⟩ `;`

⟨*structure-var-decl*⟩ ::= ⟨*id*⟩ `(` * `)*` ⟨*id*⟩

⟨*decl*⟩ ::= ⟨*var-decl*⟩
| ⟨*array-decl*⟩
| ⟨*structure-var-decl*⟩

⟨*decl-without-initialiser*⟩ ::= ⟨*type*⟩ `( * )*` ⟨*id*⟩
| ⟨*array-decl*⟩
| ⟨*structure-var-decl*⟩

⟨*parameter-list*⟩ ::= ⟨*decl-without-initialiser*⟩ `,` ⟨*parameter-list*⟩
| ⟨*decl-without-initialiser*⟩

⟨*parameter-list-opt*⟩ ::= ⟨*parameter-list*⟩ | ϵ

⟨*stmt*⟩ ::= ⟨*block-stmt*⟩
| ⟨*selection-stmt*⟩
| ⟨*iteration-stmt*⟩
| ⟨*jump-stmt*⟩
| ⟨*decl*⟩
| ⟨*expr*⟩
| ⟨*assignment-stmt*⟩
| ⟨*primary-stmt*⟩

⟨*member-access-stmt*⟩ ::= ⟨*id*⟩ `.` ⟨*member-access-stmt*⟩
| ⟨*id*⟩ `.` ⟨*id*⟩

⟨*iteration-stmt*⟩ ::= ⟨*stmt*⟩
| `break;`
| `continue;`

⟨*block-stmt*⟩ ::= `{` ⟨*stmt*⟩`*` `}`

⟨*iteration-block-stmt*⟩ ::= `{` ⟨*iteration-stmt*⟩`*` `}`

⟨*selection-stmt*⟩ ::= `if (` ⟨*expr*⟩ `)` ⟨*block-stmt*⟩
| `if (` ⟨*expr*⟩ `)` ⟨*block-stmt*⟩ `else` ⟨*block-stmt*⟩

⟨*iteration-stmt*⟩ ::= `for (` ⟨*expr-opt*⟩ `;` ⟨*expr-opt*⟩ `;` ⟨*expr-opt*⟩ `)` ⟨*iteration-block-stmt*⟩
| `for (` ⟨*decl*⟩ `:` ⟨*symbol-stmt*⟩ `)` ⟨*iteration-block-stmt*⟩
| `while (` ⟨*expr*⟩ `)` ⟨*iteration-block-stmt*⟩
| `do` ⟨*iteration-block-stmt*⟩ `while (` ⟨*expr*⟩ `) ;`

⟨*jump-stmt*⟩ ::= `return` ⟨*expr*⟩ `? ;`

⟨*assignment-op*⟩ ::= `=`
| `*=`
| `/=`
| `%=`
| `+=`
| `-=`
| `<<=`
| `>>=`
| `&=`
| `|=`
| `^=`

⟨*expr*⟩ ::= ⟨*assignment-stmt*⟩
| ⟨*var-decl*⟩

⟨*expr-opt*⟩ ::= ⟨*expr*⟩ | ϵ

⟨*assignment-stmt*⟩ ::= ⟨*logical-or-stmt*⟩
| ⟨*logical-or-stmt*⟩ ⟨*assignment-op*⟩ ⟨*assignment-stmt*⟩

⟨*logical-or-stmt*⟩ ::= ⟨*logical-and-stmt*⟩
| ⟨*logical-and-stmt*⟩ || ⟨*logical-or-stmt*⟩

⟨*logical-and-stmt*⟩ ::= ⟨*inclusive-or-stmt*⟩
| ⟨*inclusive-or-stmt*⟩ `&&` ⟨*logical-and-stmt*⟩

⟨*inclusive-or-stmt*⟩ ::= ⟨*exclusive-or-stmt*⟩
| ⟨*exclusive-or-stmt*⟩ | ⟨*inclusive-or-stmt*⟩

⟨*exclusive-or-stmt*⟩ ::= ⟨*and-stmt*⟩
| ⟨*and-stmt*⟩ ^ ⟨*exclusive-or-stmt*⟩

⟨*and-stmt*⟩ ::= ⟨*equality-stmt*⟩
| ⟨*equality-stmt*⟩ `&` ⟨*and-stmt*⟩

⟨*equality-stmt*⟩ ::= ⟨*relational-stmt*⟩
| ⟨*relational-stmt*⟩ `==` ⟨*equality-stmt*⟩
| ⟨*relational-stmt*⟩ `!=` ⟨*equality-stmt*⟩

⟨*relational-stmt*⟩ ::= ⟨*shift-stmt*⟩
| ⟨*shift-stmt*⟩ `>` ⟨*relational-stmt*⟩
| ⟨*shift-stmt*⟩ `<` ⟨*relational-stmt*⟩
| ⟨*shift-stmt*⟩ `>=` ⟨*relational-stmt*⟩
| ⟨*shift-stmt*⟩ `<=` ⟨*relational-stmt*⟩

⟨*shift-stmt*⟩ ::= ⟨*additive-stmt*⟩
| ⟨*additive-stmt*⟩ `<<` ⟨*shift-stmt*⟩
| ⟨*additive-stmt*⟩ `>>` ⟨*shift-stmt*⟩

⟨*additive-stmt*⟩ ::= ⟨*multiplicative-stmt*⟩
| ⟨*multiplicative-stmt*⟩ `+` ⟨*additive-stmt*⟩
| ⟨*multiplicative-stmt*⟩ `-` ⟨*additive-stmt*⟩

⟨*multiplicative-stmt*⟩ ::= ⟨*prefix-unary-stmt*⟩
| ⟨*prefix-unary-stmt*⟩ `*` ⟨*multiplicative-stmt*⟩
| ⟨*prefix-unary-stmt*⟩ `/` ⟨*multiplicative-stmt*⟩
| ⟨*prefix-unary-stmt*⟩ `%` ⟨*multiplicative-stmt*⟩

⟨*prefix-unary-stmt*⟩ ::= ⟨*postfix-unary-stmt*⟩
| `++` ⟨*postfix-unary-stmt*⟩
| `--` ⟨*postfix-unary-stmt*⟩
| `*` ⟨*postfix-unary-stmt*⟩
| `&` ⟨*postfix-unary-stmt*⟩
| `!` ⟨*postfix-unary-stmt*⟩

⟨*postfix-unary-stmt*⟩ ::= ⟨*primary-stmt*⟩
| ⟨*primary-stmt*⟩ `++`
| ⟨*primary-stmt*⟩ `--`

⟨*primary-stmt*⟩ ::= ⟨*constant*⟩
| ⟨*symbol-stmt*⟩
| `(` ⟨*logical-or-stmt*⟩ `)`

⟨*symbol-stmt*⟩ ::= ⟨*function-call-stmt*⟩
| ⟨*array-access-stmt*⟩
| ⟨*member-access-stmt*⟩
| ⟨*id*⟩

⟨*array-access-stmt*⟩ ::= ⟨*id*⟩ `(` `[` ⟨*expr*⟩ `]` `)*`

⟨*function-call-arg-list*⟩ ::= ⟨*logical-or-stmt*⟩ `,` ⟨*function-call-arg-list*⟩
| ⟨*logical-or-stmt*⟩

⟨*function-call-arg-list-opt*⟩ ::= ⟨*function-call-arg-list*⟩ | ϵ

⟨*function-call-expr*⟩ ::= ⟨*id*⟩ `(` ⟨*function-call-arg-list-opt*⟩ `)`

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

Spis treści