Kompilator języka strukturalnego

David Korenchuk

August, 23, 2023

Spis treści

1 Wprowadzenie

W dniu dzisiejszym istnieje wiele języków programowania. Przyczyna na istnienie narzędzia takiego rodzaju jest taka, że człowiek myśli i mówi zdaniami. Aby móc przeprowadzić ludzkie zdania do formy, zrozumiałej do komputera, niezbędnie te zdania muszą być przeprowadzone do zestawu dużo prostszych zdań, niż w ludzkich językach.

...

2 Teoria

2.1 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.

2.2 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;
}
```

3 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.

3.1 Wyrażenia regularne

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

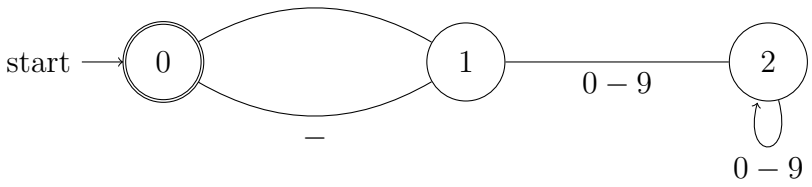
Mówimy, że wyrażenie regularne określone nad alfabetem Σ, jeżeli zachodzą następujące warunki:

- $\emptyset$ – 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

3.2 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 jezyku C. */
%}

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

%% /* Reguly w postaci wyrazen regularnych. */

/*****************************************************************/
/* Wzorzec                    | Akcja przy znalezieniu takiego wzorcu */
/*****************************************************************/
-?[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 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.

4 Analiza syntaksyczna

4.1 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).

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*⟩

4.2 Znane problemy

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:

Listing 2: Rekurencja lewa

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

4.3 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.

Listing 3: Główny węzeł AST

```
struct ast_node {
    enum ast_type  type;    /* Rozrozniamy typ wedlug 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:

Listing 4: Konkretny węzeł AST

```
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. 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
```

4.4 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 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

5.1 Analiza nieużywanych zmiennych

Podamy przykłady kodu prowadzącego do odpowiednich ostrzeżeń

Listing 5: Nieużywana zmienna

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

Listing 6: Nieużywany parametr

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

Listing 7: Nieodczytana zmienna

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

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.

5.2 Analiza typów

Podczas analizy typów należy sprawdzić, czy:

- Oba operandy wyrażenia binarnego mają ten sam typ
- Faktycznie zwracana z funkcji wartość zgadza się z jej deklaracją
- Prawidłowa ilość i typy argumentów byli przekazane do wyrażenia wywołania funkcji
- Rozmiar zadeklarowanej tablicy jest dopuszczalny
- Indeks tablicy nie wykracza poza jej rozmiar (w przypadku stałego indeksu)

Analiza typów w naszym przypadku jest dość prosta i tak samo polega na przejściu drzewa syntaksycznego.

5.2.1 Matematyczny opis systemu typów

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

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

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

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

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

$$\frac{\Gamma \vdash e_1 : float \qquad \Gamma \vdash e_2 : float}{\Gamma \vdash e_1 \oplus e_2 : float}$$

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

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 \oplus e_2 : int}$$

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

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

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

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

6 Generacja warstwy pośredniej

7 Interpreter

8 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*⟩
| `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] http://lucacardelli.name/papers/typesystems.pdf