Compiler

Vitaliy Shvets

January 2022

Spis treści

1 Wstęp

2 Teoria kompilatorów

Kompilator to program, który tłumaczy kod napisany przez człowieka na kod zrozumiały dla procesora. Kompilator jest również odpowiedzialny za wykrywanie błędów w napisanym kodzie. Kompilator można podzielić na dwie części: frontend i backend. Frontend jest odpowiedzialny za parsowanie tekstu wejściowego pod kątem gramatyki języka oraz za właściwą prezentację AST. Backend jest odpowiedzialny za optymizację i generowanie kodu dla wybranej architektury.

2.1 Lexer

Analiza leksykalna jest pierwszą fazą kompilacji kodu. Jedynym zadaniem leksera jest odczytanie kodu znak po znaku i utworzenie listy tokenów. Struktura tokena:

```
struct Token {
    Type type;
    string value;
    unsigned int line;
}
```

Oczywiście nie ma sensu tworzyć tokena dla każdego znaku w kodzie. Dlatego najpierw należy zdefiniować zasady, według których będą tworzone tokeny. Na przykład, jeśli token napotka literę, będzie odczytywał kod do momentu pojawienia się spacji lub innego znaku niż litera. Następnie zostanie utworzony token typu "identyfikator", a jego wartość będzie zawierała odczytane słowo. W ten sam sposób lekser przetwarza liczby, ciągi znaków i słowa kluczowe. Słowa kluczowe to słowa zarezerwowane w danym języku. Zazwyczaj oznaczają one początek konstrukcji językowej lub typu danych. W przeciwieństwie do ciągów znaków, dla których istnieje jeden typ tokena "string", dla każdego słowa kluczowego musi istnieć osobny typ. Dla niektórych struktur językowych, które zawierają więcej niż 2 znaki, na przykład dla "+=", lepiej jest utworzyć jeden token typu "assign_add". Optymalizacje te zmniejszają liczbę tokenów i pozwalają parserowi uniknąć wykonywania trywialnych czynności

2.2 Gramatyka

Gramatyka to zbiór reguł, które formalnie opisują składnię języka. Reguła składa się z jednego lub łańcucha znaków terminalnych i nieterminalnych. Znak terminalny to znak, dla którego nie ma reguł w gramatyce, a zatem jest gramatycznie niepodzielny. Przykład alfabetu terminalowego:

{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /}

Symbol nieterminalny to symbol odpowiadający konstrukcji gramatycznej, która może być wyrażona za pomocą reguł w postaci mniejszych konstrukcji. Innymi słowy, konstrukcja, która nie jest leksemem. Taki symbol nie ma określonego znaczenia symbolicznego. Przykład alfabetu nieterminalnego:

{wzór, znak, liczba, słowo}

Przykład reguł:

wzór -> wzór znak wzór
wzór -> liczba
wzór -> '(' wzór ')'
znak -> '+' | '-' | '/' | '*'
liczba -> '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

Ten rodzaj gramatyki nazywa się gramatyką bezkontekstową, w której reguły składają się z dwóch części:
część lewa - to dowolny symbol nieterminalny, jego znaczenie nie zależy od kontekstu, w jakim występuje.
część prawa - dowolny ciąg symboli terminalnych i nieterminalnych.
Używając tej formy notacji gramatycznej, można zdefiniować dowolną konstrukcję języka programowania. Na przykład:

```
int foo = 10;
```

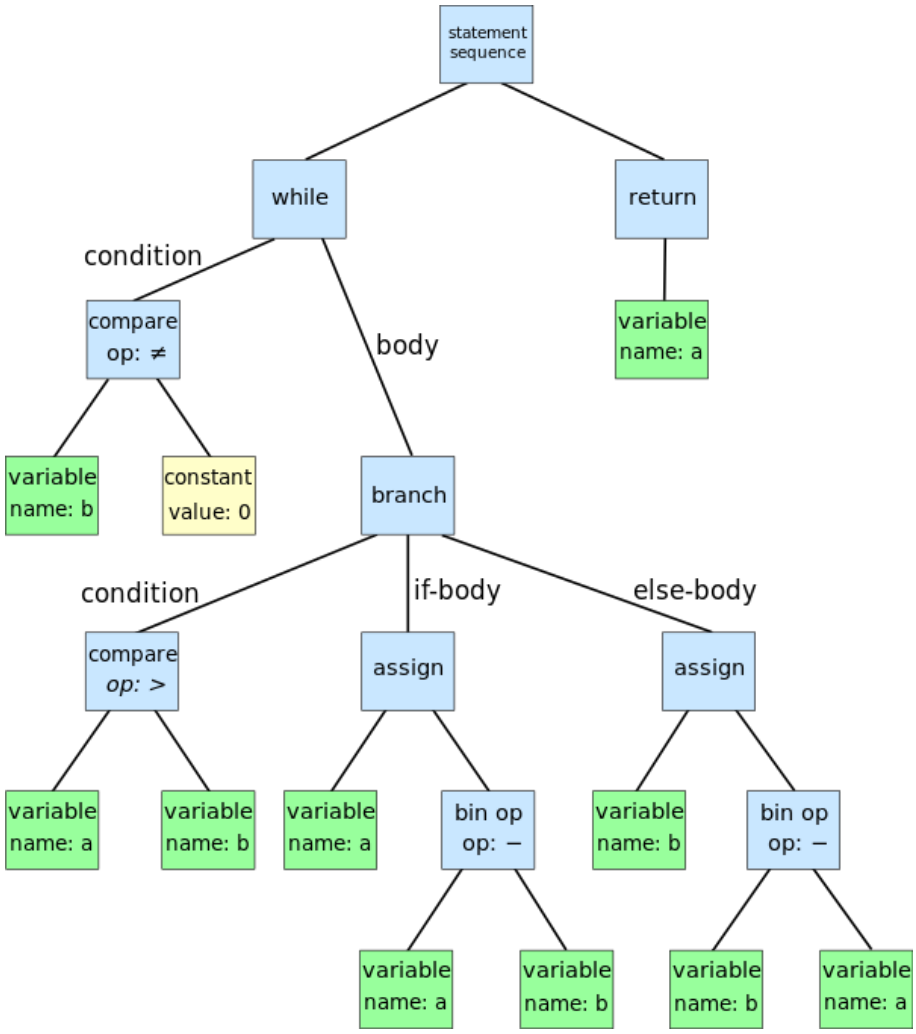
Zostanie zdefiniowana reguła deklarowania zmiennych, która wygląda następująco:

```
var_definition -> type id '=' expr ';'
```

W tym przypadku type, id, expr są innymi regułami gramatycznymi.Mogą one również składać się z innych reguł.

2.3 AST

Po zdefiniowaniu całej gramatyki języka należy przejść do fazy parsowania. Najpierw utwórz strukturę danych, która abstrakcyjnie reprezentuje kod programu. Musi ona być zgodna z wcześniej zdefiniowaną gramatyką. Struktura ta jest nazywana AST (abstract syntax tree). Drzewo wygląda następująco:



Rysunek 1: AST

W drzewie każdy węzeł jest instrukcją, która w zależności od typu ma różne liście. Liście są zmiennymi lub stałymi.

Teraz należy utworzyć to drzewo w kodzie. Najpierw zostaną utworzone klasy Stmt i Operand. Po tych dwóch klasach będą dziedziczyć wszystkie inne klasy.

```
class Stmt
{
public:
    unsigned int line;
}

class Openand
{
public:
    OpType opType;
    Prefix prefix;
    Postfix postfix;
    unsigned int line;
}
```

Klasa Operand zawiera pola prefiksów i sufiksów, takich jak '*', '', '++' itd. Typy 'Prefix', 'Sufix', 'OpType' są typami emun. Zarówno klasa Stmt, jak i klasa Operand zawierają pole line, które odpowiada za wiersz, w którym element został utworzony. Następnie zostaną utworzone klasy Id i Constant.

```
class Id : Operand
{
public:
    string name;
}

class Constant : Operand
{
public:
    ConstType constType;
    string val;
}
```

Poniżej zostanie przedstawiony przykład kilku klas dla konstrukcji programowych.

```
class Expr : public Operand {
public:
    ExprType exprType;
    Expr* left;
    Expr* right;
};

class VarDef : public Stmt {
public:
    SizeType sizeType;
    string left;
    Expr* right;
};

class If : public Stmt {
public:
    Expr* condition;
    vector<Stmt*>* stmts;
};
```

W praktyce każda z tych klas zawiera więcej pól i metod, które są potrzebne w dalszych fazach pracy kompilatora. AST nie tylko zapewniają uporządkowaną reprezentację kodu źródłowego, ale także odgrywają kluczową rolę w analizie semantycznej, podczas której kompilator sprawdza poprawność konstrukcji programowych i poprawność użycia ich elementów.

2.4 Parser

Parsowanie to druga faza części frontend.Głównym zadaniem parsera jest sprawdzanie błędów składni i generowanie AST. Parser otrzymuje ciąg tokenów od leksera i sprawdza, czy ten ciąg tokenów może być wygenerowany przez gramatykę danego języka.

Najprostszym sposobem reakcji na nieprawidłowy łańcuch tokenów wejściowych jest zakończenie pracy parsera i wydrukowanie komunikatu o błędzie. Często jednak warto znaleźć jak najwięcej błędów podczas jednej próby parsowania. Tak właśnie zachowują się parsery większości popularnych języków programowania. Komunikat o błędzie powinien zawierać jak najwięcej

informacji oraz wskazywać wiersz kodu źródłowego, w którym wystąpił błąd.

Zatem obsługa błędów parsera ma następujące zadania:

- Informowanie o błędach.
- Szybkie usuwanie błędów, aby można było kontynuować wyszukiwanie innych błędów.

Metody parsera można podzielić na metody typu top-down i bottom-up. Z nazwy można wywnioskować, że metody top-down budują drzewo od góry do dołu, natomiast metody bottom-up zaczynają się od dołu i zmierzają do korzenia.

Proces parsowania polega na odczytywaniu tokena po tokenie i wybiera ścieżkę przetwarzania tokenów w zależności od bieżącej ścieżki. Biorąc pod uwagę, że gramatyka naszego języka jest rekurencyjna, sensowne jest podjęcie próby jej rekurencyjnego przetworzenia. W tym celu należy odczytać token, a następnie przejść do następnego tokena. Na podstawie tego, jaki będzie następny token, możemy zdecydować, jaką ścieżkę musimy obrać, aby przetworzyć dane wejściowe. Może to wymagać rekursywnego wywołania funkcji, która już została wywołana. Dla każdego nieteminalnego znaku zostanie utworzona funkcja, która będzie przetwarzać tokeny zgodnie z gramatyką tego znaku. Zadaniem tych funkcji jest nie tylko sprawdzanie błędów. Tworzą one węzły drzewa AST, które zostaną wypełnione danymi podczas procesu parsowania, a następnie zwrócone z funkcji.

W kolejnych fazach kompilatora drzewo AST jest wykorzystywane do analizy i generowania programu.

2.5 Generacja kodu

Generowanie kodu jest ostatnią fazą kompilatora i obejmuje: wybór instrukcji, porządkowanie instrukcji, wybór i alokację rejestrów.

W tej fazie reprezentacja AST musi zostać przekształcona w kod niskopoziomowy wybranej wcześniej architektury. Najczęściej spotykane architektury to RISC (ang. reduced instruction set computer) i CISC (ang. complex instruction set computer). Główna różnica polega na tym, że RISC ma wiele rejestrów i instrukcje trójadresowe, natomiast CISR ma niewiele rejestrów i instrukcje dwuadresowe.

2.5.1 Wybór i porządkowanie instrukcji

Ważnym zadaniem kompilatora jest wybór najlepszych instrukcji do szybkiego wykonania programu. Każda architektura ma swój własny zestaw instrukcji, przy czym każda instrukcja ma swój czas wykonania przez procesor, co należy uwzględnić podczas generowania kodu. Bardzo ważna jest także właściwa kolejność wykonywania poleceń. Pomaga to zoptymalizować wykorzystanie rejestrów.

2.5.2 Alokacja rejestrów

Architektury procesorów mają ograniczoną liczbę rejestrów. Architektura x86-64 ma 16 rejestrów dla liczb całkowitych (RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8 - R15).Ponieważ liczba rejestrów jest ograniczona, kompilator musi umieć je poprawnie wykorzystać. Rejestry należy przydzielić w taki sposób, aby obliczenia były wykonywane prawidłowo. Oznacza to, że nie należy zapisywać nowych danych do rejestru, gdy jest on używany do innych obliczeń. Ponieważ zmiennych może być znacznie więcej niż rejestrów, w każdym momencie wykonywania programu trzeba się upewnić, w którym rejestrze znaleźć daną zmienną. Niektóre rejestry są używane do określonych celów, takich jak przekazywanie argumentów do funkcji lub zwracanie wartości z funkcji. Rejestry powinny być tak alokowane, aby w momencie, gdy trzeba przekazać wartość do funkcji, dany rejestr nie przechowywał danych z poprzednich obliczeń.

Teraz możemy przejść do samego generowania kodu. Generator pobiera węzeł AST, który będzie punktem startowym. Początek to zwykle tablica konstrukcji językowych, takich jak: definicje zmiennych, funkcji, struktur. Każdy z tych konstruktów ma swoją własną funkcję generowania kodu, która jest wywoływana w zależności od typu. I tak dla każdego węzła AST. Generator będzie odwiedzał każdy węzeł do momentu wygenerowania kodu dla każdego z nich.

Na końcu zostanie utworzony plik asemblera zawierający wygenerowany kod. Jeśli podczas generowania kodu nie wystąpią błędy, kompilator uruchomi program asemblera dla wygenerowanego pliku oraz linker, jeśli wygenerowano więcej niż jeden plik. Asembler przetłumaczy kod na instrukcje maszynowe, a linker połączy kilka plików w jeden i utworzy z nich plik wykonywalny dla systemu operacyjnego, na którym został uruchomiony kompilator.

3 Specyfikacja języka

3.1 Typy danych

Typy danych to abstrakcje nad sekwencjami binarnymi, które określają, jakie dane będą zawierać zmienne używające określonego typu danych i jak te dane będą reprezentowane w pamięci. Język zaimplementowany jest językiem statycznie typowanym. Oznacza to, że tworząc zmienne, określamy ich typy. Aby zmienić typ zmiennej, trzebna go jawnie określić. Ta metoda ma swoje wady i zalety. Zaletą tego rozwiązania jest to, że jeśli programista przypisze zmiennej inny typ danych, kompilator wskaże błąd przed uruchomieniem kodu. Ułatwia to wyszukiwanie tego typu błędów. Wadą tego rozwiązania jest to, że ponowne zdefiniowanie typów spowoduje, że kod będzie większy.

Podstawowe typy:

```
void
bool
i8
u8
i16
u16
i32
u32
string
number
```

3.2 Zmienne

Deklarowanie zmiennej jest bardzo proste - należy podać słowo kluczowe var, a następnie wpisać jej nazwę, typ i wartość. Przykład:

```
var foo: u32 = 100;
```

W ten sam sposób można zapisywać nowe wartości w zmiennych. Należy pamiętać, że typ nowej wartości musi być taki sam jak typ starej wartości.

```
var foo: u32 = 100;
foo = 200;
```

Widoczność zmiennej jest ograniczona do bloku " ". Oznacza to, że wszystkie zmienne zadeklarowane w takim bloku będą dostępne tylko w nim. Ponadto

wszystkie zmienne dostępne w obszarze, w którym utworzono blok, będą dostępne w nowym bloku. Zmienne mogą być również deklarowane poza funkcją i takie zmienne są nazywane zmiennymi globalnymi. Przykłady:

```
func main: void {
    var foo: u32 = 100;

    if foo > 50 {
        var bar: u32 = foo;
    }

    bar = 20; // error
}
```

W tym przykładzie zmienna foo jest dostępna w całej funkcji main, łącznie z blokiem if. Zapisanie wartości w pasku po wyjściu z bloku if spowoduje błąd, ponieważ nie zadeklarowano takiej zmiennej. Zmienna bar jest dostępna tylko w bloku if i nie może być używana poza nim.

3.3 Funkcje

Funkcje to małe fragmenty kodu, które mogą być ponownie wykorzystane z różnymi parametrami.

Deklaracja funkcji zaczyna się od słowa kluczowego func, po którym następuje nazwa, typ argumentów i wartość zwracana. Przykład:

```
func square(num: u32): void {
    return num * num
}

func main(): u32 {
    var foo: u32 = square(5);
}
```

Aby wywołać funkcję, wpisz jej nazwę i podaj argumenty w nawiasach W tym przykładzie wynik działania funkcji square zostanie zapisany do zmiennej foo.

3.4 Instrukcje warunkowe

Instrukcje warunkowe umożliwiają napisanie kodu, który zostanie wykonany tylko wtedy, gdy zostanie spełniony warunek określony przez programistę. Przykład:

```
var foo: u32 = 10;
if foo > 5 {
    //...
}
```

W tym przykładzie blok if zostanie wykonany, ponieważ wyrażenie foo > 5 jest prawdziwe. Warunki, które można sprawdzić:

```
>
<
>=
<=
==
!=
```

3.5 Pętle

Pętle pozwalają na wielokrotne wykonywanie tego samego fragmentu kodu bez konieczności powielania tego kodu. Przykład pętli:

```
for var i: u32 = 0; i < 10; i++ {
    //...
}
```

Pętla składa się z 3 elementów:

1. Inicjalizacja - to kod, który jest wykonywany raz przed rozpoczęciem pętli. Zwykle jest to utworzenie zmiennej, która służy jako indeks.

2. Warunek - jest to warunek, który będzie sprawdzany po każdej iteracji pętli. Dopóki warunek jest poprawny, pętla będzie kontynuowana.

3. Inkrementacja - to kod, który jest wykonywany po każdej iteracji pętli.

4 Istniejące rozwiązania realizujące podobne zadania

5 Wybrane narzędzia do implementacji

5.1 Języki

5.1.1 C++

C++ jest językiem programowania ogólnego przeznaczenia. Wysoką wydajność uzyskuje się poprzez kompilację do kodu maszynowego dla określonej architektury procesora. Ponieważ język C++ umożliwia niskopoziomowe zarządzanie pamięcią, jest używany do pisania oprogramowania niskiego poziomu, czyli systemów operacyjnych, systemów wbudowanych, sterowników, baz danych, kompilatorów, serwerów i gier.

5.2 Biblioteki

5.2.1 Flex

Flex jest narzędziem do generowania lekserów. Flex przyjmuje jako dane wejściowe plik z opisami reguł i tokenami do analizy leksykalnej. Dane wyjściowe to plik C z funkcją parsera.

5.2.2 Bison

Bison jest generatorem analizatorów składniowych, który przekształca opis gramatyki w program w języku C do parsowania tej gramatyki. Bison jest zwykle używany w połączeniu z Flex.

6 Opis implementacji

6.1 Gramatyka

```
program: definition

definition:
    | definition variable_def
```

```
    | definition function_def

variable_def: VAR ID COLON type ASSIGN or_or_expr SEMI

function_def: FUNC ID LPAREN def_args_list RPAREN COLON type LBRACE stmt

def_arg: ID COLON type

def_args_list:
    | def_arg
    | def_args_list COMMA def_arg

stmt_block:
    | stmt
    | stmt_block stmt

stmt: or_or_expr SEMI
    | assign_stmt SEMI
    | variable_def
    | if_stmt
    | for_stmt
    | return_stmt

return_stmt: RETURN or_or_expr SEMI

if_stmt: IF or_or_expr LBRACE stmt_block RBRACE

for_stmt: FOR variable_def or_or_expr SEMI or_or_expr LBRACE stmt_block

assign_stmt: operand_expr assignment_operator or_or_expr

assignment_operator: ASSIGN
    | MUL_ASSIGN
    | DIV_ASSIGN
    | MOD_ASSIGN
    | ADD_ASSIGN
    | SUB_ASSIGN
    | AND_ASSIGN
```

```
    | OR_ASSIGN

or_or_expr: and_and_expr
    | or_or_expr AND_AND and_and_expr

and_and_expr: equal_expr
    | and_and_expr AND_AND equal_expr

equal_expr: compare_expr
    | equal_expr equal_op compare_expr

equal_op: EQ
    | NEQ

compare_expr: add_expr
    | compare_expr compare_op add_expr

compare_op: LT
    | GT
    | LTEQ
    | GTEQ

add_expr: mul_expr
    | add_expr add_op mul_expr

add_op: ADD
    | SUB

mul_expr: unary_expr
    | mul_expr mul_op mul_expr
    | LPAREN or_or_expr RPAREN

mul_op: MUL
    | DIV
    | MOD

unary_expr: operand_expr
    | unary_operator unary_expr
```

```
unary_operator: MUL
     | ADD
     | SUB
     | NOT
     | AND
     | INC
     | DEC

operand_expr: primary_expr
     | postfix_expr

postfix_expr: ID
     | ID LPAREN args_expr_list RPAREN
     | postfix_expr INC
     | postfix_expr DEC

args_expr_list:
     | or_or_expr
     | args_expr_list COMMA or_or_expr

primary_expr: CHAR
     | NUMBER
     | STRING

type: BOOL
     | U8
     | I8
     | U16
     | I16
     | U32
     | I32
     | STRING_T
     | NUMBER_T
     | VOID
```

6.2 Lekser

Lekser został napisany przy użyciu narzędzia flex. Wszystkie zasady skanowania tekstu są opisane w pliku lex.l. Każda reguła to wyrażenie regularne lub słowo/symbol.

```
[0-9]*                  yylval.val = new string(yytext); return NUMBER;
L?\"(\\.|[^\\"\n])*\"   yylval.val = new string(yytext); return STRING;
'([^\r^\n^']|\\0)'      yylval.val = new string(yytext); return CHAR;
":"                     yylval.val = new string(yytext); return COLON;
";"                     yylval.val = new string(yytext); return SEMI;
"."                     yylval.val = new string(yytext); return DOT;
"var"                   yylval.val = new string(yytext); return VAR;
"func"                  yylval.val = new string(yytext); return FUNC;
"return"                yylval.val = new string(yytext); return RETURN;
[_a-zA-Z][_a-zA-Z0-9]*  yylval.val = new string(yytext); return ID;
```

6.3 AST

AST jest zbudowana przy użyciu klas i dziedziczenia w języku C++. Istnieje główna klasa bazowa Stmt, która przechowuje tylko numer linii i informacje o typie klasy. Każda klasa, która dziedziczy po Stmt, jest konstrukcją językową, taką jak definicje zmiennych, wywołania funkcji, instrukcje warunkowe itd. Ponadto klasa ExprOp dziedziczy po klasie Stmt, dzięki czemu niektóre wyrażenia mogą być używane jako konstrukcje. Ponieważ klasy Id i Call są dziedziczone po Operand, muszą mieć możliwość działania nie tylko w wyrażeniach, ale także jako niezależne konstrukcje, w tym celu stworzono klasę ExprOp.

```
var foo: u32 = 10 + bar - calc();
```

Tutaj są one niezależne.

```
func main(): u32 {
    ...
    bar++;

    calc();
    ...
}
```

6.4 Parser

Parser został napisany przy użyciu programu Bison, a gramatyka jest zdefiniowana w BNF (ang. Backus-Naur form). Na początku gramatyki znajduje się wektor definicji funkcji lub zmiennych. Ponadto dla każdej reguły jest opis akcji, które zostaną wykonane po jej znalezieniu w kodzie. Klasy AST są stworzone w bloku akcji.

```
variable_def: VAR ID COLON type ASSIGN or_or_expr SEMI
{ $$ = new VarDef($4, *$2, $6); }
```

W tym przykładzie tworzona jest klasa definicji zmiennej, w której zapisywane są typ, nazwa i jej wartość.

W niektórych regułach nie jest konieczne tworzenie nowej klasy, a jedynie użycie stałej. W tym przypadku używana jest stała, która określa, który typ jest używany

```
type: BOOL { $$ = SizeType::BOOL; }
    | U8 { $$ = SizeType::U8; }
    | I8 { $$ = SizeType::I8; }
```

6.5 Tablica symboli

Tablica symboli to struktura danych używana przez kompilator do przechowywania informacji o zmiennych i funkcjach. Tablica symboli jest zaimplementowana jako tablica mieszająca (map w C++). Kluczem jest nazwa zmiennej, a wartością klasa Symbol z informacją o typie i przesunięciu na stosie

```
map<string, Symbol> table;
```

Ponieważ zakres zmiennej jest ograniczony do bloku, w którym została ona utworzona, konieczne jest czyszczenie tablicy za każdym razem, gdy kod kończy wykonywanie bloku { }. W tym celu stworzona jest klasa Scope. Do przechowywania tablic znaków używana jest struktura stosu (stack w C++). Na początku wykonywania kodu klasa Scope tworzy nową, pustą tablicę znaków. Ponieważ zmienne mogą być deklarowane poza funkcją, następnym razem, gdy trzeba będzie utworzyć nową tablicę, należy ją rozszerzyć za pomocą metody extend().Ta metoda skopiuje bieżącą tablicę i zapisze ją jako nową na stosie. Zmienne zostaną teraz dodane do nowej tabeli. Ta

metoda pozwala kontrolować zmienne, które muszą zostać usunięte po wykonaniu bloku, w którym zostały zadeklarowane. Stworzona jest także tabela funkcji. Przechowuje ona informacje o liczbie i typach argumentów oraz typie zwracanej wartości.

6.6 Alokacja rejestrów

Klasa Register służy do przydzielania rejestrów. Posiada kilka metod, które przydzielają rejestr o wymaganym rozmiarze, oraz metodę, która zwalnia rejestr i pozwala na jego ponowne wykorzystanie.

```
int alloc8l();
int alloc8h();
int alloc16();
int alloc32();
void free(int r);
```

Informacje o wolnych i zajętych rejestrach są przechowywane w tablicę typów bool. Każdy element odpowiada za inny rejestr, zaczynający się od eax. Tablica ta wygląda następująco:

```
bool reg[22] = {
        1,  1,  1,  1,
//      eax ax  ah  al
        1,  1,  1,  1,
//      ebx bx  bh  bl
        1,  1,  1,  1,
//      ecx cx  ch  cl
        1,  1,  1,  1,
//      edx dx  dh  dl
        1,  1,  1,
//      esp sp  spl
        1,  1,  1,
//      ebp bx  bph
    };
```

Jeśli spojrzeć na tę tablicę jako na tablicę dwuwymiarową, można zauważyć, że pierwsza kolumna zawiera rejestry o rozmiarze 32 bitów, druga kolumna to część 16-bitowa, a trzecia i czwarta to części 8-bitowe. Algorytm alokacji rejestrów jest bardzo prosty - wykonuje pętlę do momentu znalezienia wolnego rejestru. Pętla rozpoczyna się z przesunięciem zależnym od żądanego

rozmiaru rejestru.

```
int Register::alloc16()
{
    for (int i = 1; i < 16; i += 4)
    {
        if(reg[i])
        {
            reg[i] = 0;
            return i;
        }
    }
}

int Register::alloc32()
{
    for (int i = 0; i < 16; i += 4)
    {
        if(reg[i])
        {
            reg[i] = 0;
            return i;
        }
    }
}
```

6.7 System typów

6.8 Generacja kodu

7 Podsumowanie

Literatura

[1] *Dick Grune.* Modern Compiler Design 2nd edition.

[2] *Alfred Aho.* Compilers: Principles, Techniques, and Tools 2nd Edition.

[3] *Bjarne Stroustrup.* The C++ Programming Language.