Schemat Działania Kompilatora

Mateusz Pieszczek

1 Wstęp

Definicja 1 ***Kompilator*** *to program (dokładniej procesor języka) który potrafi przetłumaczyć program z jednego języka (źródłowego) do innego (docelowego). Dodatkowo kompilator w trakcie translacji powinien zwracać błędy syntaktyczne i semantyczne znajdujące się w programie źródłowym.*

Najczęstszym powodem użycia kompilatorów jest uzyskanie kod w **języku maszynowym**, czyli takim który może być wykonany przez użytkownika. Taki docelowy kod zawiera już bezpośrednie polecenia np. do procesora.

Czasem gdy mamy do czynienia z językami wyższego poziomu, taki program jest kompilowany do języka niższego poziomu np. C, a dopiero potem do języka maszynowego.

Warto jeszcze wspomnieć o bardzo podobnej kategorii programów jaką są **interpretery**. Różnica polega na tym, że interpreter zamiast tworzyć kod źródłowy, bezpośrednio wykonuje polecenia zdefiniowane w programie źródłowym. Wykonanie takiego programu jest zazwyczaj dłuższe od wykonania skompilowanego programu do języka maszynowego. Zaletą jest przystępniejsze wykrywanie błędów.

Przykład 2 ***Językami kompilowanymi*** *są na przykład języki C, C++, Go, Haskell, Pascal, Fortran, Ada.* ***Językami interpretowanymi*** *są z kolei Python (formalnie jest hybrydą), R, Bash, PHP, JavaScript. Ciekawą grupą są* ***języki hybrydowe*** *jak Java, czy C#.*

Program napisany w języku Java jest najpierw kompilowany do kodu bajtowego (ang. bytecode). Ten jest następnie interpretowany przez maszynę wirtualną. Zaletą tego rozwiązania jest to ze tak skompilowany kod może być użyty na innym urządzeniu niezależnie od tego z jakiej platformy korzysta, jeśli tylko użytkownik po drugiej stronie również posiada maszynę wirtualną. C# działa podobnie. Korzysta on z infrastruktury .NET, choć w przeciwieństwie do Javy, technicznie może być on skompilowany bezpośrednio do kodu maszynowego.

Przykładowo taki kod w języku Java (https://pl.wikipedia.org/wiki/Kod_bajtowy)

```
outer:
for (int i = 2; i < 1000; i++) {
    for (int j = 2; j < i; j++) {
        if (i % j == 0)
            continue outer;
    }
    System.out.println (i);
}
```

zostaje podczas kompilacji przekształcony do następującego kodu bajtowego

```
 0:  iconst_2
 1:  istore_1
 2:  iload_1
 3:  sipush 1000
 6:  if_icmpge     44
 9:  iconst_2
 10: istore_2
 11: iload_2
 12: iload_1
 13: if_icmpge     31
 16: iload_1
 17: iload_2
 18: irem
 19: ifne   25
 22: goto   38
 25: iinc   2, 1
 28: goto   11
 31: getstatic     #84; //Field java/lang/System.out:Ljava/io/PrintStream;
 34: iload_1
 35: invokevirtual #85; //Method java/io/PrintStream.println:(I)V
 38: iinc   1, 1
 41: goto   2
 44: return
```

Do utworzenia wykonywalnego programu często używamy też dodatkowych programów. Przykładowy proces tworzenia i uruchomienia przez użytkownika programu może wyglądać następująco

1. **Preprocesor** - prostszy program, który wyszukuje i wykonuje dyrektywy preprocesora. Zazwyczaj polegają one na prostej podmianie tesktu. Typowym poleceniem w języku C jest na przykład #include który wkleja do naszego kodu źródłowego tekst innego pliku,
2. **Kompilator**,
3. **Assembler** - kompilator języka assembly,

4. **Linker** - program który łączy pliki obiektowe stworzone przez kopilator w jeden plik wykonywalny,
5. **Loader** - przekazuje treść programu do pamięci i przygotowuje je do wykonania.

2 Struktura kompilatora

Kompilację programu można podzielić na 2 główne części, analizę oraz syntezę.

Analiza ma na celu podzielenie kodu źródłowego na części składowe. Wewnątrz składowych tworzona jest struktura gramatyczna. Proces ten można przyrównać do analizy zdania języka polskiego, w której podzielilibyśmy je na zdania składowe oraz dodali etykiety do słów takie jak podmiot, orzeczenie, okolicznik. Należy jedynie pamiętać przy tym porównaniu, że człowiek analizując zdanie języka naturalnego instynktownie wykonuje takie operacje jak np. podział tekstu na zdania, słowa. To co otrzymuje kompilator to ciąg znaków i każdy proces należy skrupulatnie zaimplementować. Nawet tak prozaiczny jak podział tekstu na słowa.

Następnie możemy złożyć te składowe do struktury składniowej, która zawiera opis gramatyki poszczególnych zdań. Na niej łatwiej możemy sprawdzić czy istnieją błędy syntaktyczne, o istnieniu których zawiadomimy użytkownika. Jeśli analiza zakończy się bez znalezienia błędu, to jej produktem powinna być **tablica symboli** zawierająca informacje o nazwach(np. zmiennych, funkcji) używanych przez użytkownika oraz jakaś struktura danych opisująca gramatykę kodu.

Synteza polega na stworzeniu docelowego kodu. Wykonujemy ją już z założeniem poprawności, dzięki wcześniej wykonanej analizie. Posiadamy również na tym etapie dodatkowe informacje o gramatyce i użytym nazewnictwie dzięki tablicy symboli.

Rys. 1 pokazuje wszystkie kroki kompilacji wraz z przekazywanym wynikiem każdego procesu. Przejdziemy teraz krótko po tych etapach.

2.1 Tabela symboli

W całym procesie kompilacji korzysta się z **Tabeli Symboli**. Jej najważniejszym celem jest przechowywanie informacji o wykorzystywanych przez użytkownika nazwach. W tabeli zawarte mogą być różne informację od typu, zarezerwowanej pamięci jeśli wpis opisuje zmienną, do liczby i rodzaju argumentów jeżeli wpis opisuje jakąś funkcję użytkownika. Powodem stosowania tej tabeli jest przyśpieszenie kompilatora. Z informacji zawartych w tabeli symboli korzystamy na każdym etapie kompilacji.

Rys. 1 Schemat pracy kompilatora

2.2 Analiza Leksykalna

Analiza leksykalna jest pierwszym etapem przygotowania naszego tekstu. Ma ona na celu pogrupować nasz ciąg znaków na pojedyncze słowa i operatory. Takie pojedyncze słowa nazywamy **leksemami**, bądź jednostkami leksykalnymi. Następnie każdemu leksemowi przyporządkowuje się jego rodzaj oraz jeśli trzeba jego indeks w tabeli symboli. Taka para jest wysyłana jako **token** w strumieniu danych do następnego etapu. Przykładowymi rodzajami leksemów są

- **identyfikatory** - nazwy zmiennych
- **etykiety** - nazwy funkcji
- **operatory** - np. +,-,*,=
- **literały** - np. liczba, wartość logiczna, znak, string

Przykład 3 *Przeprowadźmy analizę leksykalną na poniższym fragmencie kodu.*

```
predkosc = poczatkowa + czas * 10
```

Lexemy które w wyniku analizy otrzymamy to:

- ***predkosc*** *- identyfikator, zapiszemy go jako <id,1>, i przyporządkujemy mu pierwszą pozycję w tabeli symboli*
- **=** *- operator, zapiszemy jako <=>*
- ***poczatkowa*** *- identyfikator, zapiszemy go jako <id,2>, i przyporządkujemy mu drugą pozycję w tabeli symboli*
- **+** *- operator, zapiszemy jako <+>*
- ***czas*** *- identyfikator, zapiszemy go jako <id,3>, i przyporządkujemy mu trzecią pozycję w tabeli symboli*
- ***** *- operator, zapiszemy jako <*>*
- ***10*** *- literał liczbowy, zapiszemy go jako <10> (można by oczywiście zapisać tą liczbę w tablicy symboli, ale w naszym prostym przypadku załóżmy uproszczoną sytuację gdzie, literały zapisujemy tak jak są)*

Finalnie otrzymamy ciąg znaków, który przekażemy do Analizatora Składni oraz uzupełnioną tablicę symboli o zmienne i etykiety, które znaleźliśmy w kodzie.

```
<id,1> <=> <id,2> <+> <id,3> <*> <10>
```

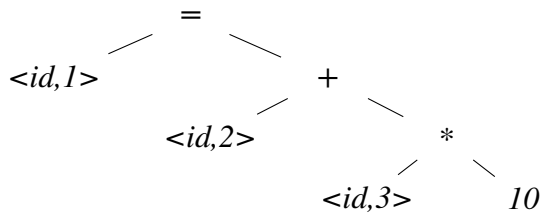
2.3 Analiza Składniowa

Analiza składni/syntaktyki otrzymuje na wejście z poprzedniego etapu dobrze opisane słowa naszego programu w strumieniu tokenów. Zadaniem tego etapu jest powiązanie tych słów strukturą gramatyczną. Typową wykorzystywaną strukturą jest drzewo składniowe, gdzie właściwe węzły są operacjami, a w dzieciach takiego węzła znajdują się argumenty. Znanym nam już analizatorem składni jest algorytm zamieniający zapis infiksowy na ONP. Nie wiemy jeszcze czy wygenerowane w tym etapie zdania są poprawne, jest to rola kolejnego etapu.

Przykład 4 *Dla przykładu z poprzedniej sekcji*

```
predkosc = poczatkowa + czas * 10
```

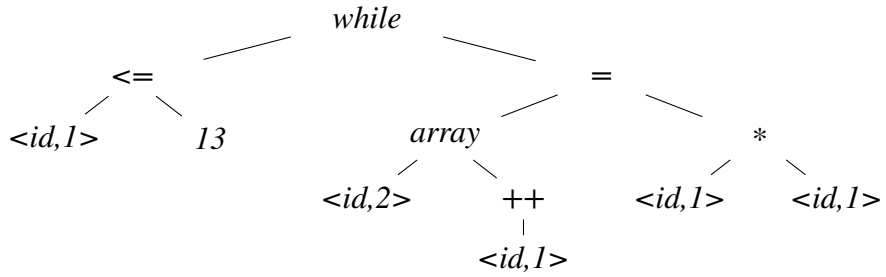
drzewo składniowe tego zdania miało by postać jak na poniższym rysunku.



Przykład 5 *Dla bardziej złożonego kodu*

```
while(i <= 13){
    tablica[++i] = i*i
}
```

takie drzewo mogło by wyglądać tak



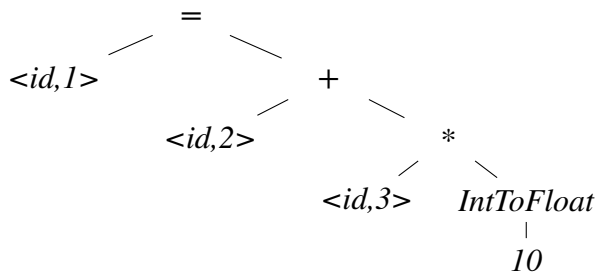
2.4 Analiza Semantyczna

Podczas tego etapu przeprowadzamy **analizę błędów** w oparciu o wszystkie informacje, które dotychczas zgromadziliśmy. Jednym z częstych zadań tej analizy jest sprawdzenie **typowania**. Wszystkie operatory, funkcje posiadają pewne wymogi dotyczące argumentów. Przykładem może być wymóg, aby indeks tablicy był liczbą naturalną, albo chcemy aby instrukcja warunkowa **if**, czy pętla **while** miały na wejściu wartość logiczną, itd. Na tym etapie może dojść do pewnych modyfikacji. Nasz zdefiniowany język źródłowy może na przykład posiadać standardową definicje dodawania liczb zmiennoprzecinkowych. Ale chcielibyśmy również posiadać, dodawanie liczba całkowitych i zmiennoprzecinkowych, które na wyjściu zwracałoby wartość zmiennoprzecinkową. Operacja ta jest intuicyjna, ale należy pamiętać że sposób zapisu tych typów jest inny i musimy rzutować jeden z argumentów na typ drugiego argumentu.

Przykład 6 *Dla naszego przykładu*

```
predkosc = poczatkowa + czas * 10
```

możemy założyć, że nasze zmienne predkosc, poczatkowa i czas są float-ami. Stałą 10 wprowadziliśmy jako integer. Możemy zaadresować tą niezgodność typów dodając rzutowanie w naszym drzewie



Jeśli w podczas tego etapu odnajdziemy sytuację, nie dopuszczalną w gramatyce języka wejściowego, np. indeksowanie tablicy float-em, to kompilator powinien zakończyć swoje działanie niepowodzeniem informując użytkownika w precyzyjny sposób, jakiego typu wystąpił błąd i jeśli to możliwe, gdzie on wystąpił w kodzie źródłowym.

2.5 Pośrednie generowanie kodu

Przychodzimy teraz do etapów syntezy kompilatora. Specyfika tych etapów mocno zależy od języka docelowego. W ogólności podczas generowania pośredniego kodu, chcemy stworzyć kod w języku przypominający docelowy, który po optymalizacji pozwoli na przetłumaczenie w szybki sposób na język docelowy. Najlepiej będzie to widać na naszym przykładzie.

Przykład 7 *Załóżmy, że naszym językiem docelowym jest assembly. Podczas optymalizacji chcemy operować w pewnej abstrakcji np. nie chcemy jeszcze bezpośrednio deklarować, których rejestrów użyjemy. Wtedy dla naszego kodu obliczającego prędkość możemy stworzyć następujący kod pośredni.*

```
t1 = inttofloat(10)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

gdzie zmienne t1, t2, t3 reprezentują pewne abstrakcyjne rejestry pamięci. Jeśli docelowo tworzymy kod assembly, to nasz kod pośredni będzie się spełniał zasadę trzy-adresowych instrukcji. Dzięki niej zazwyczaj jedna instrukcja pośredniego kodu przekłada się na jedną instrukcję kodu assembly. Dzięki temu optymalizacja dokonana na kodzie pośrednim będzie optymalizowała też kod docelowy.

Uwaga 8 *Ten etap jest przygotowaniem przed następnym, czyli optymalizacją kodu, która jak można się domyśleć jest opcjonalna. Jednakże rzadko znajdziemy kompilatory bez nawet minimalnej formy optymalizacji, dlatego krótko wspomnimy o oby tych etapach.*

2.6 Optymalizacja Kodu

Mając już kod w języku pośredni, możemy dokonać optymalizacji. Jest to najbardziej czasochłonny proces całej kompilacji. Zazwyczaj sprowadza się to do usuwania instrukcji które w języku docelowym są redundantne. Przykładem mogą być pętle lub instrukcje warunkowe bez instrukcji w swoim ciele. Możliwości redukcji jest wiele, a jedynym ograniczeniem jest to aby nasz program po optymalizacji produkował identyczny wynik jak program wejściowy. Tutaj projektant kompilatora musi dokonać decyzji co pozostawia w odpowiedzialności użytkownika. W wyniku nadmiernej opiekuńczości złożona optymalizacja mogła by być bardziej czasochłonna niż zauważenie błędu przez użytkownika podczas testów programu. Taka optymalizacja mogła by nie mieć sensu.

Uwaga 9 *Optymalizację możemy wykonać też po wygenerowaniu kodu docelowego, czyli na sam koniec kompilacji. Proces jest podobny. Takie rozdzielenie ma w niektórych sytuacjach sens. Mając postać finalną kodu docelowego możemy skupić się na innych aspektach, np. alokacji pamięci czy doborze rejestrów w przypadku assembly.*

Przykład 10 *Weźmy nasz kod z poprzedniej sekcji. Pierwszą rzecz jaką widać to niepotrzebne mapowanie wartości 10 na float. Moglibyśmy przecież od razu przekazać do rejestru t1 wartość 10.0. Kolejnym elementem jest to że korzystamy z rejestrów t1 oraz t3 tylko po to żeby przekazać wartość do kolejnych instrukcji podczas gdy moglibyśmy wartości przekazać bezpośrednio. Po takich poprawkach nasz kod wyglądałby tak*

```
t1 = id3 * 10.0
id1 = id2 + t1
```

Przykład 11 *W przypadku skoków które powstaną podczas tłumaczenia pętli taki fragment kodu*

```
goto ABC
label ABC
```

moglibyśmy całkowicie usunąć. Prawdopodobnie źródłowo został dodany dla przejrzystości kodu (bądź nie. Nie możemy zakładać racjonalności użytkownika.), ale tutaj jedynie niepotrzebnie wydłuża nam kod o 2 linijki.

2.7 Generowanie Kodu

Przechodzimy do ostatniego etapu jakim jest generowanie docelowego kodu. Zazwyczaj na tym etapie będziemy decydować o wyborze rejestrów i alokacji pamięci na używane

zmienne. W zależności od tego jaki język tłumaczymy ten etap może się diametralnie różnić. Chcąc na przykład stworzyć kompilator języka C wiemy, że alokacja pamięci jest w pewnym stopniu pozostawiona użytkownikowi. Dla języków wyższego poziomu wiele operacji jest automatyzowanych, co oznacza, że praca kompilatora zabiera więcej czasu.

Przykład 12 *Przekształcając kod na assembly z poprzedniego przykładu otrzymamy*

```
LDF    R2,    id3
MULF   R2,    R2, #10.0
LDF    R1,    id2
ADDF   R1,    R1, R2
STF    id1,   R1
```

2.8 Uwagi Ogólne

2.8.1 Grupowanie etapów

Powyższy podział pracy kompilatory jest teoretycznym schematem. W praktyce możemy połączyć pewne etapy, zamienić etapy kolejnością (w szczególności optymalizację). Na przykład możemy wyobrazić sobie proces który równocześnie interpretuje słowa i w tej samej pętli wstawia je do pewnej struktury składniowej.

2.8.2 Narzędzia tworzenia kompilatorów

Współcześnie istnieje wiele narzędzi automatyzujące pewne etapy tworzenia kompilatorów. Są to narzędzia pozwalające projektować kompilatory w wyższym poziomie abstrakcji. Na takiej samej zasadzie nie zajmujemy się alokacją pamięci w językach wyższego poziomu. Przykładowymi narzędziami są:

- **Generator Parserów** - na podstawie gramatycznego opisu języka tworzy analizator składniowy
- **Generator Skanerów** - na podstawie wyrażeń regularnych(regex-ów) opisujących tokeny tworzy analizator leksykalny
- **Silnik Tłumaczenia Składni** - tworzy zasady przechodzenia po drzewie składni w celu generowania pośredniego kodu
- **Generator Generatorów Kodu** - na podstawie reguł translacji tworzy odpowiedni generator kodu
- **Silnik Analizy Przepływu-Danych** - gromadzi informację o przepływie danych w kompilowanym programie. Często wykorzystywany jest przy optymalizacji danych.

Takich narzędzi jest oczywiście znacznie więcej, a to są jedynie przykłady, które czytelnik po przeczytaniu poprzednich sekcji powinien intuicyjnie zrozumieć.

3 Podsumowanie

Dziękuję za przeczytanie/wysłuchanie mojego tekstu. Osoby które w między czasie może zaciekawiłem, odsyłam do książki 'Compilers: Principles, Techniques, and Tools' https://web.archive.org/web/20110513112656/http://dragonbook.stanford.edu/ z której głównie korzystałem. Osoby chcące zobaczyć faktyczny kod kompilatora mogę odesłać do kodu źródłowego gcc https://github.com/gcc-mirror/gcc/tree/master/gcc. Można pobrać repozytorium i poeksperymentować.