

Prezentacja – kompilatory

22.03.2018

Program, który czyta kod napisany w jednym języku (*źródłowym*) i tłumaczy go na równoważny kod w drugim języku (*wynikowym*)

- Analiza leksykalna
- Analiza składniowa
- Analiza semantyczna
- *Generacja kodu pośredniego*
- *Optymalizacja*
- Generacja kodu

Kod źródłowy

```
// Ustaw wartosc a  
var a = 4 + 0.2
```

⇒

Tokeny

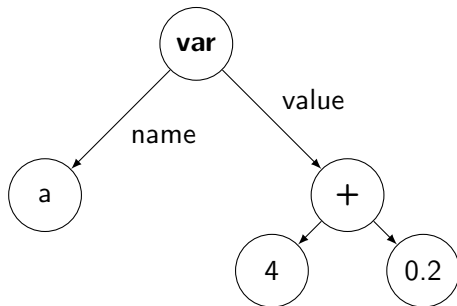
```
keyword(var),  
id(a),  
operator(=),  
integer(4),  
operator(+),  
real(0.2)
```

Tokeny

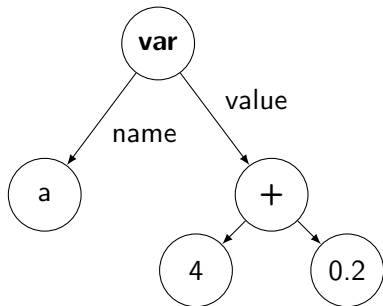
keyword(var),
id(a),
operator(=),
integer(4),
operator(+),
real(0.2)

⇒

AST

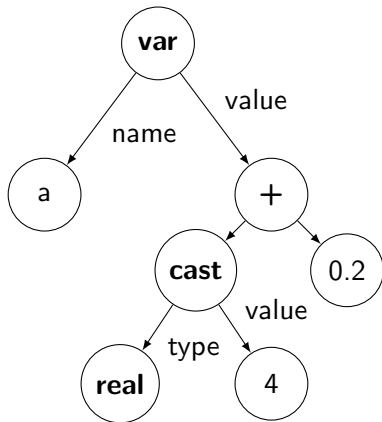


AST

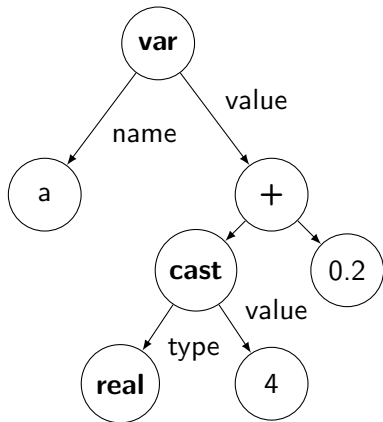


\Rightarrow

AST'



AST'



⇒ ... ⇒

Kod wynikowy

```
.LC0:  
    .long 1082549862  
_start:  
    movss xmm0, .LC0[rip]  
    movss -4[rbp], xmm0  
    mov eax, 0  
    ret
```

Notacja infiksowa \rightarrow Notacja postfiksowa

- Notacja infiksowa: $e_1 + e_2$
- Notacja postfiksowa (odwrotna notacja polska): $e_1 e_2 +$
 - Przykłady:
 - $1 + 2 * 3 \Rightarrow 1\ 2\ 3\ *\ +$
 - $(1 + 2) * 3 \Rightarrow 1\ 2\ +\ 3\ *$
 - Nie wymaga nawiasów
 - Łatwo obliczać wartość wyrażenia (implementacja na stosie)
 - Niestety nieczytelna dla ludzi :(

Napiszmy kompilator!

Definicja składni v1 (Składnia abstrakcyjna)

wyrażenie \rightarrow wyrażenie + wyrażenie |
 wyrażenie * wyrażenie |
 (wyrażenie) | liczba

Definicja składni v1 (Składnia abstrakcyjna)

wyrażenie \rightarrow wyrażenie + wyrażenie |
wyrażenie * wyrażenie |
(wyrażenie) | liczba

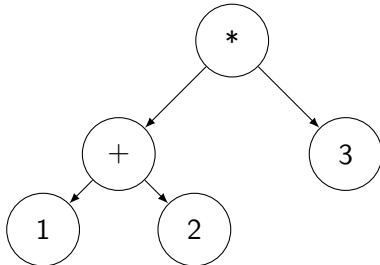
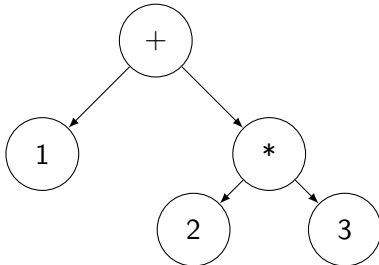
Jaki jest problem?

Definicja składni v1 (Składnia abstrakcyjna)

wyrażenie \rightarrow wyrażenie + wyrażenie |
wyrażenie * wyrażenie |
(wyrażenie) | liczba

Jaki jest problem?

1 + 2 * 3



Definicja składni v2 (Składnia konkretna)

wyrażenie \rightarrow składnik | składnik + wyrażenie

składnik \rightarrow czynnik | czynnik * składnik

czynnik \rightarrow (wyrażenie) | liczba

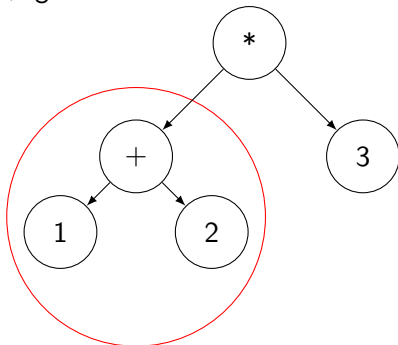
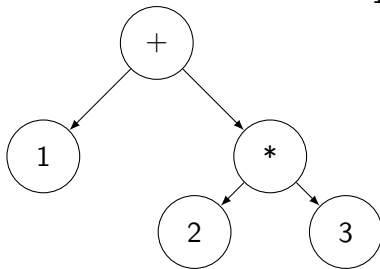
Definicja składni v2 (Składnia konkretna)

wyrażenie \rightarrow składnik | składnik + wyrażenie

składnik \rightarrow czynnik | czynnik * składnik

czynnik \rightarrow (wyrażenie) | liczba

1 + 2 * 3



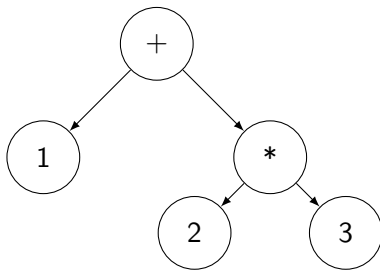
Definicja składni v2 (Składnia konkretna)

wyrażenie \rightarrow składnik | składnik + wyrażenie

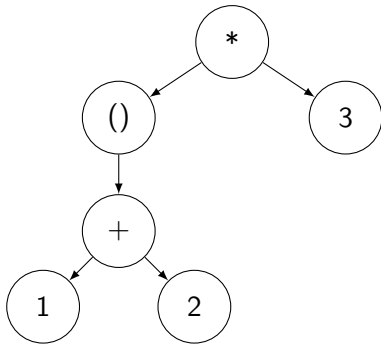
składnik \rightarrow czynnik | czynnik * składnik

czynnik \rightarrow (wyrażenie) | liczba

1 + 2 * 3



(1 + 2) * 3



Dygresja: składnia abstrakcyjna vs składnia konkretna

Składnia abstrakcyjna

- Prostsza
- Każde drzewo wyprowadzenia wyznacza jednoznacznie program
- Potencjalnie wiele drzew wyprowadzeń dla danego programu

Składnia konkretna

- Definiowanie wymaga więcej wysiłku
- Drzewo wyprowadzenia może zawierać nadmiarowe węzły
- Zwykle spełnia dodatkowe własności, umożliwiającą wydajną analizę składniową (np. jednoznaczność, LL, LR)

- `obecny()` – zwraca obecny znak
- `nastepny()` – wczytuje kolejny znak z wejścia
- `wypisz()`, `wypisz_liczbe()` – emituje ciąg znaków/liczbę
- `error()` – sygnalizuje błąd i kończy program

Lekser (analiza leksykalna)

```
int liczba() {  
    int wynik = 0;  
    if (!isdigit(obecny())) error();  
    while (isdigit(obecny())) {  
        wynik = wynik*10 + (obecny() - '0');  
        nastepny();  
    }  
    return wynik;  
}
```

Lekser (analiza leksykalna)

```
int liczba() {  
    int wynik = 0;  
    if (!isdigit(obecny())) error();  
    while (isdigit(obecny())) {  
        wynik = wynik*10 + (obecny() - '0');  
        nastepny();  
    }  
    return wynik;  
}  
  
bool probuj_znak(char c) {  
    if (obecny() != c) return false;  
    nastepny();  
    return true;  
}
```

Lekser (analiza leksykalna)

```
int liczba() {
    int wynik = 0;
    if (!isdigit(obecny())) error();
    while (isdigit(obecny())) {
        wynik = wynik*10 + (obecny() - '0');
        nastepny();
    }
    return wynik;
}

bool probuj_znak(char c) {
    if (obecny() != c) return false;
    nastepny();
    return true;
}

void znak(char c) {
    if (!probuj_znak(c)) error();
}
```

Parser (analiza składowa) + generator kodu

wyrażenie \rightarrow składnik | składnik + wyrażenie

```
void wyrażenie() {  
    składnik();  
    if (próbuj_znak('+')) {  
        wyrażenie();  
        wypisz("+ ");  
    }  
}
```

Parser (analiza składowa) + generator kodu

składnik \rightarrow czynnik | czynnik * składnik

```
void skladnik() {  
    czynnik();  
    if (probuj_znak('*')) {  
        skladnik();  
        wypisz("* ");  
    }  
}
```

Parser (analiza składniowa) + generator kodu

czynnik \rightarrow (wyrażenie) | liczba

```
void czynnik() {  
    if (próbuj_znak('(')) {  
        wyrażenie();  
        znak(')');  
    } else {  
        int n = liczba();  
        wypisz_liczbe(n);  
    }  
}
```

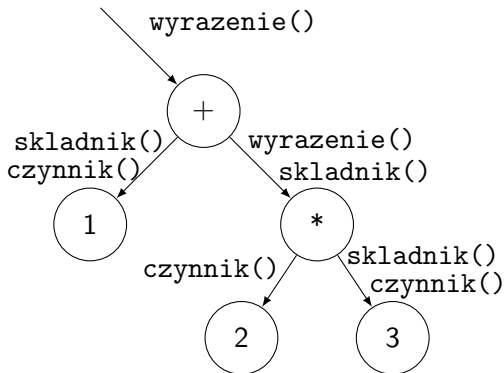
Brakujące AST

Ok, tylko gdzie tu jest drzewo AST?

Brakujące AST

Ok, tylko gdzie tu jest drzewo AST?
Jest konstruowane na bieżąco!

1 + 2 * 3



Demo – kompilator do ONP

Bardziej skompilowane gramatyki

- Powyższa gramatyka była bardzo prosta – nigdy nie potrzebowaliśmy więcej niż następnego znaku
- Prawdziwe gramatyki są bardziej skomplikowane
- Ogólne metody analizy języków bezkontekstowych – $O(n^3)$
- Języki $LL(k)$ – mogą być sparsowane zstępująco używając co najwyżej najbliższych k znaków
- Języki $LR(k)$ – mogą być sparsowane wstępująco używając co najwyżej najbliższych k znaków

Automatyczne generowanie parserów

- lex – generator lekserów
- yacc (yet another compiler compiler) – generator parserów
- bison – nowsza wersja programu yacc

Notacja infksowa → Notacja postfiksowa raz jeszcze

lex

```
[0-9]+ {yylval=atoi(yytext); return LICZBA;}  
\n      return 0;  
      return *yytext;
```

yacc

```
%token LICZBA  
%left '+' '*'  
E:    E '+' E {printf("+ ");}  
      | E '*' E {printf("* ");}  
      | '(' E ')'  
      | LICZBA {printf("%d ", yyval);}
```

- Bardzo zależna od języka
- Wykrywanie konstrukcji poprawnych składniowo, ale nie mających sensu w semantyce języka
- Obliczanie różnych cech poszczególnych węzłów i zapisywanie ich w drzewie
- Sprawdzanie typów
- Modyfikacja drzewa AST dla operacji implicite (np. rzutowanie w językach słabo typowanych)

Generowanie kodu pośredniego

- Kod dla pewnej abstrakcyjnej maszyny, bliższej maszynie docelowej, ale ukrywającej jej szczegóły
- Ułatwia zmianę języka docelowego (np. na inną architekturę procesora)
- Wnioskowanie i optymalizacje często są łatwiejsze niż na drzewie AST
- Umożliwia wydajniejszą interpretację (uproszczone parsowanie i kompilacja)

Rodzaje kodu pośredniego

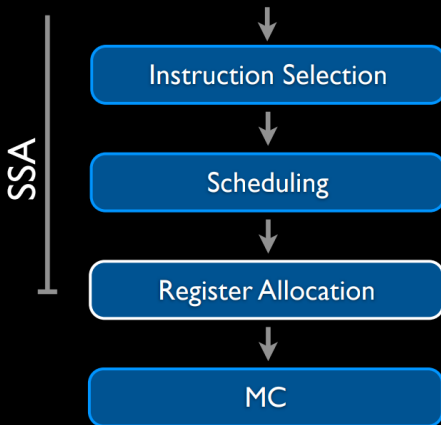
- Kod dla maszyny rejestrowej
 - LLVM
 - nasm
- Kod dla maszyny stosowej
 - bajtkod .NET – CIL
 - bajtkod JVM
 - bajtkod CPythona
 - bajtkod WebAssembly

- Silnie typowany kod dla maszyny rejestrowej z funkcjami
- Standardowy backend dla różnych języków (C, C++, Haskell, Fortran)
- Kod trójadresowy: większość instrukcji jest postaci
wynik = operator typ arg1 arg2
- SSA – static single assignment, na dowolny rejestr można przypisać wartość tylko raz

LLVM – przykład (ze smurfa [4])

```
define i32 @fact(i32 %n) {  
    %c0 = icmp eq i32 %n, 0  
    br i1 %c0, label %L0, label %L1  
L0:  
    ret i32 1  
L1:  
    %i1 = sub i32 %n, 1  
    %i2 = call i32 @fact(i32 %i1)  
    %i3 = mul i32 %n, %i2  
    ret i32 %i3  
}
```

Register Allocation in LLVM



Bibliografia

- 1 A.V. Aho, R. Sethi, J.D. Ullman, Kompilatory. Reguły, metody i narzędzia
- 2 http://wazniak.mimuw.edu.pl/index.php?title=SW_wyk%C5%82ad_2_-_Slajd2
- 3 http://wazniak.mimuw.edu.pl/index.php?title=SW_wyk%C5%82ad_2_-_Slajd3
- 4 <http://smurf.mimuw.edu.pl/node/797>
- 5 https://llvm.org/devmtg/2011-11/01esen_RegisterAllocation.pdf
- 6 <https://llvm.org/devmtg/2016-09/slides/Absar-SchedulingInOrder.pdf>