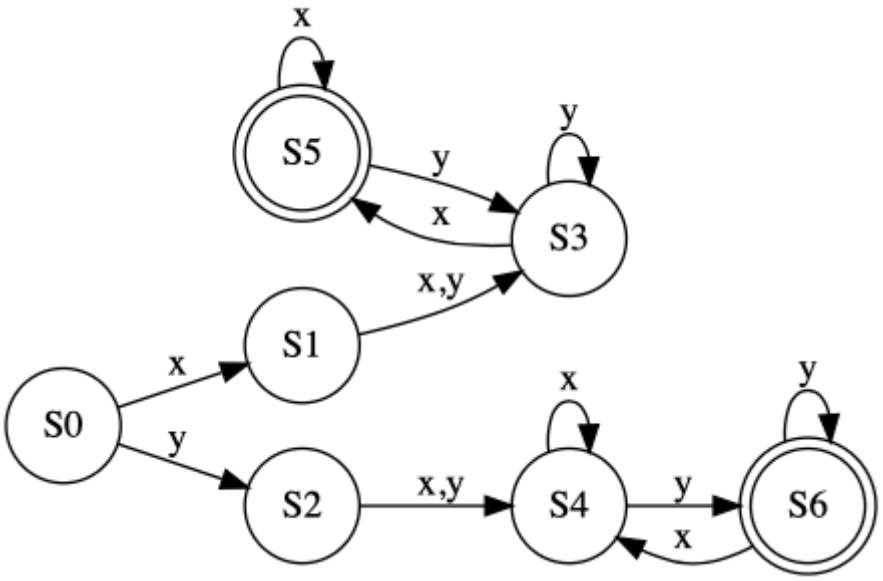
Programski Prevodioci - Priprema za ispit

Napomena: *Priloženi zadaci su samo polazna tačka za spremanje ispita i ne obuhvataju sve oblasti objašnjene na predavanjima.*

CFG Developer: https://web.stanford.edu/class/archive/cs/cs103/cs103.1156/tools/cfg
LLVM: https://drive.google.com/file/d/1UE45xzbdkS_l5hPfzFM6dIR05ZqITz7w/view

---

1. Grafički predstaviti deterministički konačni automat koji prepoznaje reči nad azbukom {x, y}, takve da imaju bar tri znaka, a počinju i završavaju se istim znakom. Formalno specificirati regularnu gramatiku G=(∑, N, S, P) koja generiše ovakve reči.



Iz DKA se direktno može izvesti sledeća gramatika:

∑ = {x, y}
N = {S0, S1, S2, S3, S4, S5, S6}
S = {S0}
P = {
S0 → x S1 | y S2
S1 → x S3 | y S3
S3 → y S3 | x S5
S5 → x S5 | y S3 | ε
S2 → x S4 | y S4
S4 → x S4 | y S6
S6 → x S6 | x S4 | ε
}

Daljim zaključivanjem se može izvesti sledeća gramatika:

∑ = {x, y}
N = {REC, NIZ, SLOVO}
S = {REC}
P = {
REC → x NIZ x | y NIZ y
NIZ → x NIZ | y NIZ | SLOVO
SLOVO → x | y
}

2. Objasniti tipičnu ulogu nedeterminističkog konačnog automata u projektovanju kompajlera.

Nedeterministički konačni automat (NKA) u projektovanju kompajlera ima ulogu u konstrukciji determinističkog konačnog automata (DKA) na osnovu regularnih izraza. NKA je pogodan jer se šablonski konstruiše iz konkretnog regularnog izraza. Konverzija NKA u DKA je poželjna zato što NKA zahteva kompleksniju implementaciju nego DKA zbog sledećih karakteristika:

- NKA može imati više pomeranja za jedan ulaz
- NKA ima ε-pomeranje
- NKA sadrži skup stanja koji se može reprezentovati kao jedno stanje u DKA

Leksička specifikacija → Regularni izrazi → NKA → DKA → Tabela stanja → Izvršni kod

3. Treba li regularnim izrazima definisati lekseme koje ne pripadaju jeziku? Obrazloži i ilustrovati primerima.

Poželjno je definisati lekseme koje ne pripadaju jeziku zato što te lekseme predstavljaju greške. Programeru će značiti detaljna informacija o grešci koja može nagovestiti potencijalnu izmenu koju treba uraditi kako bi proces kompajliranja bio uspešan. Pre definisanja specifičnih regularnih izraza, potrebno je evidentirati koje se greške najčešće prijavljuju od strane kompajlera.

Primeri:

NAZIV_NE_MOZE_POCETI_CIFROM
→ [0-9]+[a-zA-Z]+

NAZIV_NE_MOZE_SADRZATI_SPECIJALNE_ZNAKOVE
→ [a-zA-Z]*SPECIJALNI_ZNAK[a-zA-Z]*, gde je SPECIJALNI_ZNAK → !|@|#|$|%|...

4. Za jezik definisan gramatikom napisati kod silaznog rekurzivnog parsiranja. Kod komentarisati objašnjenjima.

E → T | T-E
T → INT | INT*T | (E)

```
// pokazivač se postavlja na sledeći token i proverava se da li je taj token očekivan
bool term(TOKEN token) { return *next++ == token; }

// prva alternativa prve produkcije
bool e1() { return t(); }

// druga alternativa prve produkcije
// funkcija treba vratiti “true” ako je tačna svaka konkatenacija pa se koristi znak “&&”
bool e2() { return t() && term(MINUS) && e(); }

// čuva se pokazivač na trenutni token i ispituju se alternative prve produkcije
// funkcija treba vratiti “true” ako je tačna jedna alternativa pa se koristi znak “||”
// budući da je “E” startni simbol, pozivanjem ove fukcije otpočinje parsiranje
bool e() {
        TOKEN* save = next;
        return (next = save, e1()) || (next = save, e2());
}

// prva alternativa druge produkcije
bool t1() { return term(INT); }

// druga alternativa druge produkcije
bool t2() { return term(INT) && term(TIMES) && t(); }

// treća alternativa druge produkcije
bool t3() { return term(LPAREN) && e() && term(RPAREN); }

// čuva se pokazivač na trenutni token i ispituju se alternative druge produkcije
bool t() {
        TOKEN* save = next;
        return (next = save, t1()) || (next = save, t2()) || (next = save, t3());
}
```

5. Na osnovu zadatog segmenta koda kompajlera izvedi zaključak o kakvoj pretrazi je reč. Detaljno opiši ulogu ovakvih pretraga.

```
void begin_scope() {
        stack_push(NULL);
}

id* lookup(char* key) {
        unsigned i = hash(key) % SIZE;
        bucket* b;
        for (b = table[i]; b; b = b->next)
                if (!strcmp(b->key, key))
                        return b->binding;
```

```
        return NULL;
}

void end_scope() {
        while(1) {
                char* s = stack_pop();
                if (s == NULL)
                        break;
                delete(s);
                free(s);
        }
}
```

Priloženi kod vrši pretragu tabele simbola na način na koji se ne narušava princip postojanja više okruženja promenljivih (scope-ova) tako što se svako okruženje stavlja na vrh steka pri ulasku u blok instrukcija i skida sa steka pri izlasku iz bloka instrukcija. Ovakva pretraga je neophodna za interpretiranje koda zato što je interpreteru neophodna informacija o vrednosti koju određeni simbol trenutno poseduje. Stek okruženja nije neophodan ako bi se proces interpretacije izvršavao nad kodom kod koga nisu prisutne funkcije i ugneždeni blokovi instrukcija.

6. Iz kojih osnovnih delova se sastoji LLVM?

Sastoji se iz sledećih delova:

1. Front-end deo koji vrši leksičku, sintaksnu i semantičku analizu i generiše međureprezentaciju
2. Middle-end deo koji vrši arhitektonsko nezavisne optimizacije nad međureprezentacijom
3. Back-end deo koji vrši arhitektonsko zavisne optimizacije nad međureprezentacijom i generiše ciljni kod

7. Koji su osnovni zadaci srednjeg dela LLVM-a?

Srednji deo (Middle-end) spaja prednji i zadnji deo kompajlera, što omogućava LLVM-u da obrađuje više programskih jezika i da generiše mašinski kod za više platformi. Nad međureprezentacijom se u ovom delu izvršavaju mašinski nezavisne optimizacije, npr. optimizacija petlji pomoću komponente polly.

8. Iz kojih entiteta se sastoji hijerarhijska struktura LLVM međureprezentacije?

Sastoji se iz sledećih entieta:

1. Modul - definiše sadržaj datoteke u kojoj se nalazi međukod
2. Funkcija - konceptualno odgovara funkciji iz jezika C
3. Osnovni blok - niz instrukcija koji ima samo jednu tačku ulaza i izlaza
4. Instrukcije