Zadaća 6 iz predmeta Dizajn kompajlera

Problem

Potrebno je napraviti jednostavni, interpretirani programski jezik. Za izradu zadataka koristiti flex, bison, kao i metod evaluacije programa pomoću stabla, kao na laboratijskim vježbama 8.

Ovaj jezik treba da podržava sljedeće:

- jedini tip podatka je *integer*,
- aritmetičke operacije nad integerima (+, -, *, /),
- logičke operacije (==, !=, <, >),
- definiciju i čitanje varijabli,
- kontrolu toka pomoću *if-else* naredbi,
- petlje pomoću *while* naredbe,
- definiciju i pozivanje funkcija.

U prilogu zadaće se nalazi šablonski kod od kojeg možete nastaviti sa izradom zadaće.

Početna gramatika

Kod u prilogu sadrži evaluator sa sljedećom gramatikom:

```
program = program statement
        | epsilon

statement = expression ';'
          | print expression ';'
          ;
expression = expression + expression
           | expression - expression
           | '(' expression ')'
           ;
```

Evaluacija svakog *statementa* se izvršava u glavnom pravilu za program, koja vrti programsku petlju. Za razliku od primjera na vježbama, ovaj evaluator treba da prikaže ispis korisniku samo kad korisnik unese `print` komandu.

U semantičkim akcijama se alociraju i uvezuju čvorovi, koji čine stablo za izvršenje. Čvorovi su definirani u fileu `expression_tree.hpp`.

Zadatak 1 - implementirati operacije koje nedostaju

Trenutni kod podržava sabiranje i oduzimanje. Potrebno je implementirati ostale aritmetičko logičke operacije. Postoji i mogućnost korištenja zagrada kako bi precizno odredili redoslijed izvršenja navedenih operacija.

Vaš zadatak:

- Implementirati dodatne tokene potrebne za ove operacije.
- Obratiti pažnju na prioritet i asocijativnost tokena.
- Za svaku operaciju dodati alternativno pravilo u expression.
- Za svaku operaciju napraviti tip čvora koji izvršava tu operaciju, po uzoru na postojeće čvorove za sabiranje i oduzimanje.
- U semantičkoj akciji za ova pravila napraviti novi čvor.

Konačni cilj je da program može izvršavati kod tipa:

```
print 1 == 1;
> 1
print 1 != 1
> 0
print 1 < 5;
> 1
print 1 > 5;
> 0
print 10 / 5 ;
> 2
print 2 * 5;
> 10
```

Zadatak 2 - varijable, dodjela i čitanje vrijednosti

Sve varijable žive u strukturi Environment, koja je definirana u fileu 'environment.hpp'. Varijable predstavljaju mapu koja čuva ime varijable i vrijednost koja je trenutno asocirana sa tom varijablom.

Novo pravilo u nizu alternativa za expression bi imalo sljedeću strukturu:

```
expression : ID '=' expression
```

Ovo pravilo upisuje u mapu varijabli pod ključem ID i vrijednosti evaluiranog expressiona. Obratiti pažnju da se ovo dešava tek pri evaluaciji statementa. Za ovu namjenu već imate implementiranu `AssignmentNode` strukturu.

Vaš zadatak:

- Dodati pravilo u expression alternative,
- U semantičkoj akciji za to pravilo alocirati čvor tipa `AssignmentNode`.
- Pogledati njenu implementaciju, potpis konstruktora.

Pravilo za čitanje vrijednosti varijable bi imalo strukturu:

```
expression : ID
```

Ovo pravilo čita vrijednost pod ključem ID iz mape varijabli. Za ovu namjenu imate napravljenu strukturu `VarijableNode`.

Vaš zadatak:

- Dodati pravilo u expression alternative,
- U semantičkoj akciji za to pravilo alocirati čvor tipa `VariableNode`.
- Pogledati njenu implementaciju, potpis konstruktora.

Konačni cilj je da program može izvršavati kod tipa:

```
a = 10;
print a;
> 10
b = a + 10;
print b;
> 20
```

Zadatak 3 - if i else statementi

Do sada su sva proširenja gramatike bila vezana za expression. Glavna razlika između expressiona i statementa je što expressioni imaju povratnu vrijednost, dok statement nema. Statement može biti expression, kod kog je odlučeno da se povratna vrijednost zanemari. Budući da if i if-else naredbe nemaju smislenu povratnu vrijednost u ovom dizajnu, njihova produkcijska pravila će biti smještena kao statement.

Pravilo za if naredbu bi bilo:

```
statement : IF '(' expression ')' statement
```

Čvor koji modelira ponašanje ove naredbe naziva se IfNode i njegovu implementaciju možete pronaći u kodu.

Vaš zadatak:

- Dodati pravilo u statement alternative,
- U semantičkoj akciji za to pravilo alocirati čvor tipa `IfNode`.
- Pogledati njenu implementaciju, potpis konstruktora.

Pravilo za if-else naredbu bi izgledalo ovako:

```
statement : IF '(' expression ')' statement ELSE statement
```

Ova komanda je proširenje if-a. Za nju je potrebno osmisliti i napraviti čvor koji modelira ponašanje if-else komande.

Vaš zadatak:

- Dodati pravilo u statement alternative,
- Napraviti čvor tipa `IfElseNode`, po uzoru na `IfNode`.
- U semantičkoj akciji za to pravilo alocirati čvor tipa `IfElseNode`.

Cilj je da program može izvršavati if i else naredbe, u sljedećem formatu:

```
if (1) print 10;
> 10
if (0) print 10;
if (1) print 10; else print 20;
> 10
if (0) print 10; else print 20;
> 20
```

Zadatak 4 - while petlja

While je jedna od osnovnih petlji i njena sintaksa ima sljedeću strukturu:

```
statement : WHILE '(' expression ')' '{' statement_list '}'
```

Kako ni petlja nemaju smislenu povratnu vrijednost, ovo pravilo se dodaje kako statement. Ovo je prvi put da se susrećemo sa produkcijskim pravilom statement_list. Njegov atribut predstavlja pokazivač na vektor statementa koji se nalaze u tijelu jednog bloka. Implementaciju ovog pravila možete pronaći u lang.y fileu i obratiti pažnju na način kako se taj vektor alocira, te kako se elementi ubacuju.

Vaš zadatak:

- Dodati pravilo u statement alternative,
- U semantičkoj akciji za to pravilo alocirati čvor tipa `WhileNode`.
- Pošto konstruktor ovog čvora očekuje vektor, a ne pokazivač na vektor, potrebno je dereferencirati vrijednost koju dobijemo is statement_list pravila.
- Implementirati evaluate metod za `WhileNode`, na način da dok se god uslov evaluira na tačno, izvrši se tijelo petlje

Cilj je da se sljedeći dok uspješno izvrši:

```
a = 4;
while (a) {
  print a;
  a = a - 1;
```

```
}
> 4
> 3
> 2
> 1
```

Zadatak 5 - definicija i poziv funkcije

Sintaksa definicije funkcije bi bila sljedeća:

```
statement : ID '(' id_list ')' '{' statement_list '}'
```

dok bi za pozive funkcija imali sljedeću sintaksu:

```
expression : ID '(' expression_list ')'
```

U `expression_tree.hpp` fileu se nalaze 3 tipa čvora koji su namjenjeni za pozive funkcija:

- `FunctionNode` koji predstavlja samu funkciju i njeno tijelo. Evaluacija ovog čvora je zapravo evaluacija same funkcije. Ovaj čvor živi u `Environment` varijabli, u mapi funkcija. Ovaj čvor se ne pravi direktno u prokukcijskim pravilima, nego ga pravi i registruje `FunctionDefinitionNode`.
- `FunctionDefinitionNode` je čvor koji predstavlja definiciju funkcije. Njegova uloga je da kreira `FunctionNode` i postavi ga u `Environment` mapu funkcija, čineći ga time dostupnim za pozive.
- `FunctionCallNode` je čvor koji predstavlja poziv funkcije. Njegova uloga je dohvatanje `FunctionNode`-a iz mape unutar Environmenta, evaluacija izraza koji su parametri pozivu funkcije, i pripremanje tih parametara za tijelo funkcije.

Vaš zadatak:

- Dodati gornja dva pravila u sintaksu programa.
- Alocirati odgovarajuće čvorove.
- Implementirati metod evaluate za `FunctionCallNode`. On treba da provjeri da li postoji funkcija sa datim imenom i dohvati pokazivač na funkciju. U tom momentu, dostupni su vektor ExpressionNode-ova koji prestavlja parametre pozivu funkcije, i vektor imena argumenata koje funkcija očekuje. Ovaj metod treba da evaluira ExpressionNode-ove i njihov rezultat spremi u mapu varijabli pod imenom koje je parametar funkcije. Na primjer ako imamo funkciju `foo(a,b,c)` i pozovemo je na način `foo(1, 2, 3)`, potrebno je u environment varijable ubaciti key-value parove {a, 1}, {b,2}, {c, 3}.
- Na kraju treba evaluirati samu funkciju.

Dodatno, potrebno je implementirati jedan od ponuđenih dodatnih zadataka.

Dodatni zadatak - environment stack

Trenutno ako bi se izvršio dio koda, imali bi sljedeći ispis:

```
foo(a) { print a; }
foo(5);
> 5
print a;
> 5
```

Iz toga vidimo da se varijable u environmentu ne resetuju, odnosno da nemamo čišćenje stacka nakon poziva funkcije.

Kako bi ovo implementirali, unutar strukture `Environment` možemo koristiti strukturu stack. Mapa koja pohranjuje varijable u tom trenutku bi postala `std::stack<std::map<std::string, int>>`. Kada pokušamo dohvatiti neku varijablu, indeksiramo samo vrh stacka. Prije ulaska u funkciju, na ovaj stack gurnemo praznu mapu. Neposredno nakon izvršenja tijela funkcije, sa vrha stacka maknemo tu mapu. Time smo osigurali da funkcija ima vidljivost samo svojih parametara i lokalnih varijabli.

Gornji kod bi tada postao:

```
foo(a) { print a; }
foo(5);
> 5
print a;
> 0
```

Dodatni zadatak - povratna vrijednost funkcije

Potrebno je osmisliti i implementirati način da se iz funkcije vrati vrijednost natrag u pozivajuću funkciju. U sintaksu programa bi ubacili pravilo:

```
RETURN expression;
```

Jedan od načina bi bio da u mapu varijabli, pod ključem `return` spremimo povratnu vrijednost, te je dohvatimo nakon izvršenja bloka funkcije.

Predaja zadaće

Potrebno je napraviti zaseban direktorij sa imenima ime_prezime_dk2020 u kome je rješenje problema i priložiti ga zadaći. Unutar direktorija se nalaze sve potrebne datoteke za kompajliranje programa. Problem mora imati svoj Makefile Program će koristit standardni ulaz i izlaz.

Svaki pokušaj prepisivanja i plagijata će biti oštro sankcionisan od strane predmetnog asistenta.