DOKUMENTACIJA

Programski prevodioci - predmetni zadatak

Osnovni podaci

Broj indeksa	Ime i prezime	Tema
SW14/2019	Ana Anđelić	Pokazivači

Korišćeni alati

Naziv	Verzija
hipsim	1.2
bison	3.8.2
flex	2.6.4
gcc	11.2.0

Evidencija implementiranog dela

Za ovaj zadatak je urađena sintaksna i leksička analiza uvođenjem tipova int* i unsigned*, kao i operatora * i &.

Semantička analiza je urađena za provere šta se može dereferencirati i referencirati, takođe i dodela vrednosti neodgovarajućeg tipa.

Generisanje koda je urađeno dodavanjem instrukcija LOAD i UNLOAD. LOAD instrukcija ima sintaksu LOAD op1, op2, gde se u operator op2 upisuje adresa operatora op1. UNLOAD instrukcija ima sintaksu UNLOAD op1, op2, gde se u operator op2 upisuje vrednost sa adrese koju sadrži operator op1.

Nije bilo moguće implementirati u potpunosti simulator ovih instrukcija, zbog zbunjujećeg čuvanja vrednosti trenutnog operanda u memoriji, i generalno jer kod simulatora nije napravljen da bude lako proširiv, stoga sam implementirala leksičku, sintaksnu, semantičku analizu i generisanje koda za nizove i for petlje.

For petlje su kompletno implementirane sa sintaksnom, semantickom analizom i generisanjem koda, sa inkrement i dekrement operatorima.

Nizovi su kompletno implementirani sa sintaksnom i semantičkom analizom i generisanjem koda. Nije implementirano deklarisanje veličine niza promenljivom, samo literalom.

Detalji implementacije

1. Pokazivači

1. 1. Sintaksna, semantička i leksička analiza i generisanje koda

Prvi korak je bio dodavanje novih tokena. Dodati su tokeni za operatore * i &, kao i tipovi podataka int* i unsigned* u datoteci micko.l. Nakon toga su dodata polja PINT i PUINT za tip u enumeraciji types u datoteci defs.h.

Sledeći korak sintaksne analize je bio da se korišćenjem ovih tokena dopuni sintaksa mini C jezika. Kod izraza exp je dodata sintaksna analiza u vidu _AMPERSAND _ID i _ASTERISK _ID koji redom, služe za dobavljanje adrese i dobavljanje vrednosti sa neke adrese. Oni su opisani na slici 1 i 2. Prilikom generisanja operacije dodele adrese operatorom & se proverava da li je promenljiva definisana tokenom _ID deklarisana, zatim se generiše kod za ovu naredbu i sintaksa je LOAD op1, op2, gde je zamišljeno da se adresa operatora op1 smešta u operator op2.

```
// & _ID
| _AMPERSAND _ID
    {
      int idx = lookup_symbol($2, VAR|PAR);
      if($$ == NO_INDEX)
        err("'%s' undeclared", $2);

      $$ = take_reg();
      code("\n\t\tLOAD\t");
      gen_sym_name(idx);
      code(",");
      gen_sym_name($$);
      set_atr2($$, idx);
      set_type($$, get_type(idx)+2);
    }
```

Slika 1 - Sintaksa i semantika operatora &

Prilikom generisanja operacije dodele vrednosti sa adrese adrese operatorom * se proverava da li je promenljiva definisana tokenom _ID deklarisana, zatim da li je odgovarajućeg tipa, jer nije moguće dereferencirati bilo šta što nije pokazivvač, zatim se generiše kod za ovu naredbu i sintaksa je UNLOAD op1, op2, gde je zamišljeno da se vrednost na adresi koju sadrži operator op1 smešta u operator op2.

```
// * _ID
| _ASTERISK _ID
    {
      int idx = lookup_symbol($2, VAR|PAR);
      if($$ == NO_INDEX)
        err("'%s' undeclared", $2);

      if(get_type(idx) != PINT && get_type(idx) != PUINT)
        err("operator can't be dereferenced");

      $$ = take_reg();
      code("\n\t\tUNLOAD\t");
      gen_sym_name(idx);
      code(",");
      gen_sym_name($$);
      set_type($$, get_type(idx)-2);
    }
```

Slika 2 - Sintaksa i semantika operatora *

1. 2. Izmena simulatora

Glavni problem izrade ovog zadatka je izmena simulatora. S obzirom da su dodate dve nove instrukcije, potrebno je podržati ih u simulatoru hipsim. Prvi korak dopune ovog simulatora je dodavanje prepoznavanja tokena _LOAD i _UNLOAD u hipsim.l datoteci, kao i dodavanje enumeracije o tipu instrukcije, odnosno dodavanje INS_LOAD i INS_UNLOAD u enumeraciju insntrukcija u defs.h datoteci.

Sledeća izmena je dopuna hipsim.y datoteke prepoznavanjem instrukcija LOAD i UNLOAD dopunom arithmetic izraza prikazano na slici 3.

```
185
186     |   _LOAD input _COMMA output
187         {
188             insert_source("\t\t\tLOAD %s, %s", $2, $4);
189             insert_code(INS_LOAD, NO_TYPE, yylineno);
190             free($2); free($4);
191         }
192
193     |   _UNLOAD input _COMMA output
194         {
195             insert_source("\t\t\tUNLOAD %s, %s", $2, $4);
196             insert_code(INS_UNLOAD, NO_TYPE, yylineno);
197             free($2); free($4);
198         }
199     ;
```

Slika 3 - Dopuna arithmetic izraza

Što se tiče samog koda simulatora, bilo je potrebno dopuniti switch case narednu funkcije run_once prikazano na slici 4.

```
551         case INS_LOAD:
552             set_operand(i->op[1], get_operand(i->op[0]));
553             processor.pc++;
554             break;
555         case INS_UNLOAD:
556             set_operand(i->op[1], get_operand(i->op[0]));
557             processor.pc++;
558             break;
```

Slika 4 - Dopuna switch case naredbe funkcije run_once

Ovde nailazimo na problem. Kako čuvati adresu operanda?. Operand struktura sadrži polje data koje definiše podatak koji sadrži taj operand. Nije moguće dobaviti lokaciju ove promenljive zato što je temporary, kako postoje samo 3 operatora koja se nalaze u sistemu u jednom trenutku jer svaka instrukcija može imati 3 operatora. Sledi slika 5 koja prikazuje funkcije get_operand i set_operand. Adresa se u ovom slučaju može odnositi na bilo koji od ovih 6 switch case opcija. Ideja je bila da se dobavljaju i smeštaju adrese samo varijabli na steku, kao na primer u našem asembleru -8(%14), međutim nisam znala koji od ovih 6 enumeracija označava promenljivu na steku.

```
332 //vraća vrednost operanda
333 word get_operand(Operand op) {
334     switch (op.kind) {
335         case OP_REGISTER:
336             return processor.reg[op.reg];
337         case OP_INDIRECT:
338             return *getmem(processor.reg[op.reg]);
339         case OP_INDEX:
340             return *getmem(processor.reg[op.reg]+op.data);
341         case OP_CONSTANT:
342             return op.data;
343         case OP_ADDRESS:
344             return op.data;
345         case OP_DATA:
346             return *getmem(symtab[op.data].address);
347         default: {
348             simerror("get_operand fatal error"); // ne bi trebalo da se desi
349         }
350     }
351 }
352
353 //postavlja vrednost operanda
354 void set_operand(Operand op, word data) {
355     switch (op.kind) {
356         case OP_REGISTER:
357             processor.reg[op.reg] = data;
358             break;
359         case OP_INDIRECT:
360             *getmem(processor.reg[op.reg]) = data;
361             break;
362         case OP_INDEX:
363             *getmem(processor.reg[op.reg]+op.data) = data;
364             break;
365         case OP_ADDRESS:    //možda ne treba...
366             op.data = data;
367             break;
368         case OP_DATA:
369             *getmem(symtab[op.data].address) = data;
370             break;
371         default: {
372             simerror("set_operand fatal error"); // ne bi trebalo da se desi
373         }
```

Slika 5 - Funkcije get_operand i set_operand

2. For petlja

Što se tiče leksičke analize, dodati su samo operatori ++ i --, kao i prepoznavanje ključne reči for. Sintaksa, semantika i generisanje koda su bili vrlo jednostavni za implementiranje. Prvo je dodat novi iskaz pod nazivom for_statement u statement, zatim je implementiran i sam for_statement prikazan na slici 6.

```
217
218 for_statement
219   : _FOR
220       {
221         loop_num++;
222         $<i>$ = loop_num;
223       }
224
225     _LPAREN _ID _ASSIGN literal
226       {
227         int idx = lookup_symbol($4, VAR|PAR);
228         if(idx == NO_INDEX)
229           err("'%s' undeclared", $4);
230
231         gen_mov($6,idx);
232
233         code("\n@for_cmp_%d:", $<i>2);
234       }
235
236     _SEMICOLON rel_exp
237       {
238         code("\n\t\t%s\t@for_end_%d", opp_jumps[$9], $<i>2);
239       }
240
241     _SEMICOLON _ID _INC _RPAREN statement
242       {
243         int idx = lookup_symbol($12, VAR|PAR);
244         if(idx == NO_INDEX)
245           err("'%s' undeclared", $12);
246         if(idx != lookup_symbol($4, VAR|PAR))
247           err("Iterators don't match");
248
249         if ($13 == INC) {
250           if(get_type(idx) == INT)
251             code("\n\t\tADDS\t");
252           else
253             code("\n\t\tADDU\t");
254         }
255         else {
256           if(get_type(idx) == INT)
257             code("\n\t\tSUBS\t");
258           else
259             code("\n\t\tSUBU\t");
260         }
261
262         gen_sym_name(idx);
263         code(",$1,");
264         gen_sym_name(idx);
265         code("\n\t\tJMP \t@for_cmp_%d", $<i>2);
266
267         code("\n@for_end_%d:", $<i>2);
268       }
269
```

Slika 6 - Sintaksa, semantika i generisanje koda for petlje

Prvo je podržana sintaksa for iskaza koja glasi for (id = literal ; rel_exp ; id ++) statement. Da bismo podržali više petlji jedne nakon drugih, kao i jedne unutar drugih

i izbegli problem više labela sa istim nazivom, dodeljujemo for petlji jedinstven identifikator. Svaki put kada se kreira nova for petlja, dobiće identifikator u vidu trenutnog broja petlji koji se skladišti u privremenom tokenu $2, nakon čega se ovaj broj inkrementira. Zatim se proverava da li postoji definisana promenljiva sa nazivom koji sadrži token _ID, dobijamo grešku ukoliko ne postoji.

For petlja se sastoji iz tri dela, prvi deo je poređenje rel_exp izraza, zatim je drugi deo telo for petlje, i na kraju poslednji deo je uvećavanje ili umanjenje iteratora.

3. Nizovi

Prvo su dodati tokeni _LSB i _RSB koji redom označavaju left square bracket (“[”) i right square bracket (“]”), kao i _COMMA koji označava običan zarez. Dodata je i vrsta ARRAY koju imaju svi elementi niza.

Urađena je inicijalizacija niza na istom mestu gde se inicijalizuju promenljive. Inicijalizacija niza se može vršiti inicijalizacijom praznog niza, kao i popunjavanjem niza literalima. Prilikom inicijalizacije niza se na steku zauzima onoliko promenljivih koliko ima članova niza plus jedan koji je ispred svih i na kog svi pokazuju. Kada se niz deklariše sa previše ili premalo literala ispisuje se greška. Sintakse za inicijalizaciju niza su _TYPE [literal] _ID i _TYPE [literal] _ID = { literal_list }. Pristup elementima niza se vrši putem sintakse _ID [literal]. Na slikama 7, 8, 9, 10 i 11 je prikazan kod sintaksne i semantičke analize kao i generisanje koda za redom inicijalizaciju niza, inicijalizaciju niza listom literala, sama lista literala, dodelu vrednosti elementu niza i pristup elementu niza.

```
142   | _TYPE _LSB literal _RSB _ID _SEMICOLON
143       {
144         if(get_type($3) == UINT)
145           err("invalid index value");
146         int idx;
147         if(lookup_symbol($5, VAR|PAR|ARRAY) == NO_INDEX)
148            idx = insert_symbol($5, ARRAY, $1, ++var_num, NO_ATR);
149         else
150            err("redefinition of '%s'", $5);
151
152         int max = atoi(get_name($3));
153         if(max < 1)
154            err("index out of bounds");
155         arr_num += max;
156         for (int i = 0; i < max; i++) {
157           insert_symbol(strdup($5), ARRAY, $1, idx, i);
158         }
159       }
```

Slika 7 - Inicijalizacija niza

```
161   | _TYPE _LSB literal _RSB _ID _ASSIGN _LBRACKET
162       {
163         if(get_type($3) == UINT)
164           err("invalid index value");
165         int idx;
166         if(lookup_symbol($5, VAR|PAR|ARRAY) == NO_INDEX)
167            idx = insert_symbol($5, ARRAY, $1, ++var_num, NO_ATR);
168         else
169            err("redefinition of '%s'", $5);
170
171         int max = atoi(get_name($3));
172         if(max < 1)
173            err("index out of bounds");
174         arr_num += max;
175         array_first_elem = idx + 1;
176         array_elem_num = max;
177         for (int i = 0; i < max; i++) {
178           insert_symbol(strdup($5), ARRAY, $1, idx, i);
179         }
180         iterator = 0;
181       }
182     literal_list _RBRACKET _SEMICOLON
183       {
184         if (iterator < array_elem_num)
185            err("too few elements");
186       }
```

Slika 8 - Inicijalizacija niza listom literala

```
189 literal_list
190   : literal
191       {
192         if (get_type($1) != get_type(array_first_elem + iterator))
193           err("incompatible types");
194         gen_mov($1, array_first_elem);
195         iterator++;
196       }
197   | literal_list _COMMA literal
198       {
199         if (iterator >= array_elem_num)
200            err("too many elements");
201         else {
202           if (get_type($3) != get_type(array_first_elem + iterator))
203             err("incompatible types");
204           gen_mov($3, array_first_elem + iterator);
205         }
206         iterator++;
207       }
208   ;
```

Slika 9 - Inicijalizacija niza listom literala

```
294   | _ID _LSB literal _RSB _ASSIGN num_exp _SEMICOLON
295       {
296         if(get_type($3) == UINT)
297           err("invalid index value");
298         int idx = lookup_symbol($1, ARRAY);
299         if(idx == NO_INDEX)
300           err("invalid lvalue '%s' in assignment", $1);
301         if(get_type(idx) != get_type($6))
302           err("incompatible types in assignment");
303
304         else {
305           int offset = atoi(get_name($3));
306           int elem_index = idx - get_atr2(idx) + offset;
307
308           if(offset < 0 || offset > get_atr2(idx))
309             err("index out of range");
310           else
311             gen_mov($6, elem_index);
312         }
313       }
314   ;
```

Slika 10 - Dodela vrednosti elementu niza

```
353   | _ID _LSB literal _RSB
354       {
355         if(get_type($3) == UINT)
356           err("invalid index value");
357         int idx = lookup_symbol($1, ARRAY);
358         if(idx == NO_INDEX)
359           err("'%s' undeclared", $1);
360
361         int offset = atoi(get_name($3));
362         if(offset < 0 || offset > get_atr2(idx)) {
363           err("index out of range");
364         }
365
366         $$ = take_reg();
367         int elem_index = idx - get_atr2(idx) + offset;
368         gen_mov(elem_index, $$);
369       }
```

Slika 11 - pristup vrednosti elementa niza

Da bi se kod izgenerisao pravilno, dopunjena je funkcija gen_sym_name tako da se pravilno generiše pristup elementima niza u asemblerskom kodu, što je prikazano na slici 11.

```
if(get_kind(index) == ARRAY) {
  code("-%d(%%14)", (get_atr1(get_atr1(index)) + get_atr2(index)) * 4);
}
```

Slika 12 - Dopuna funkcije gen_sym_name

Ideje za nastavak

Trenutno pokazivači imaju urađenu sintaksnu, semantičku i leksičku analizu, dok se pri izvršavanju asemblerskog koda ponašaju kao obične promenljive tipa int ili unsigned. Ideja za nastavak bi ovde bila više udubljivanja u kod hipsim simulatora i pronalaženje načina za čuvanje adrese operatora.

For petlje su implementirane kompletno, tako da ovde nema prostora za napredak.

Nizovi funkcionišu u svakom slučaju osim kada im se deklariše dužina putem druge promenljive. Moguće je uraditi int[4] dok nije moguće uraditi int[a] gde je a prethodno definisano i dodeljena mu je neka vrednost.

Literatura

Korišteni su samo pdf-ovi sa vežbi.