Əsrin Proqramlaşdırma Dili Layihəsi

Açıq Kodlu C++ layihəsi.

Lisenziya GPL V3.

Müəllif: Əhməd Sadıxov.

Ümumi tanışlıq.

İDE

Mətn redaktoru olaraq açıq kodlu Notepad++ proqramından istifadə olunur. Redaktora əsrin kompilyatoru və icraçısı inteqrə olunub, **İcra** menyusunun **Kompilyasiya et** və **İcra et** əmrlərinə müvafiq olaraq.

Kompilyasiya Et əmri Notepad_plus.rc faylının (sətir 266) `IDM_CMPL case-inə` tikilib.

// Notepad_plus.rc

```
POPUP "&İcra"
 BEGIN
     MENUITEM "&Kompilyasiya Et...",    IDM_CMPL
     MENUITEM "&Icra Et...",    IDM_EXECUTE
 END
```

IDM_CMPL case –i NppCommands.cpp faylında çağırılır(sətir 1358). Əvvəlcə aktiv pəncərənin mətni `fileSaveAsAhm()` funksiyası ilə `"esrin_src_prg.esr"` faylında yadda saxlayır (`NppİO.cpp sətir 963`)

```
case IDM_CMPL:
{
        fileSaveAsAhm();
```

Daha sonra ShellExecute ilə `"compile_bt.cmd"` skripti işə salınır arxa fonda.

```
SHELLEXECUTEINFO ShExecInfo = {0};
ShExecInfo.cbSize = sizeof(SHELLEXECUTEINFO);
ShExecInfo.fMask = SEE_MASK_NOCLOSEPROCESS;
ShExecInfo.hwnd = NULL;
ShExecInfo.lpVerb = NULL;
ShExecInfo.lpFile = TEXT("compile_bt.cmd");
ShExecInfo.lpDirectory = TEXT(".");
ShExecInfo.lpParameters = TEXT("");
ShExecInfo.nShow = SW_HIDE;
ShExecInfo.hInstApp = NULL;
::ShellExecuteEx(&ShExecInfo);
::WaitForSingleObject(ShExecInfo.hProcess, INFINITE);
```

Kompilyator kompilyasiya nəticələrini **Netice** adlı faylda yadda saxlayır. Daha sonra `fileOpenAhm()` funksiyası (NppİO.cpp :1092) həmin faylı oxuyaraq Notepad++ redaktorunun aktiv pəncərəsinin alt hissəsində çap edir.

İcra et əmri isə İDM_EXECUTE case –inə tikilib.

```
case IDM_EXECUTE:
{
	//fileSaveAsAhm();
	system("run.cmd");

	break;
	}
```

Burada isə `"run.cmd"` skripti işə salınır və kompilyatorun generasiya etdiyi ikili faylı (prg.esr_bin) icra edir.

Kompilyator

Kompilyasiya prosesi aşağıdakı mərhələlərdən ibarətdir:

1) Mətnin tokenlər ardıcıllığına çevrilməsi.
2) Parsinq ağacının qurulması

a. Sintaksis səhvlərin yoxlanması

3) Simvollar cədvəlinin qurulması
    a. Semantik səhvlərin yoxlanılması
4) Kod generasiyası

Bütün bu işlər müvafiq olaraq aşağıdakı funksiyalar vastəsilə həyata keçirilir:

Esrin.cpp

```
tokenize("esrin_src_prg.esr");

parse();

symtab();

generate();
```

---

tokenize () funksiyası token.cpp faylında təyin olunub.

```
void tokenize(char *src){
    get_tokens(src);
     init_hash_ids();
     //print_tokens();
}
```

get_tokens () funksiyası mətni bayt-bayt oxuyaraq ondan ədələri, sətirləri, simvolları, açar sözləri, riyazi, müqaisə v.s. operatorları, mötərizələri ümumiyyətlə dilin lüğətinə daxil olan bütün sözləri ayırır və **tokens** arrayinə - cərgəsinə yerləşdirir. tokens cərgəsi struct token tiplidir

token.cpp

```
#define MAX_TOKENS 100000

struct token tokens[MAX_TOKENS];
```

esrin.h

```
struct token {
  /* identification  */
  char *tok;
```

```
  int id;
  int prs_id;
  unsigned long  hid;
  /* variable recognition  */
  int tip;
  int size;
  int ofst; /* id in vartab, ofst of parent el in case of fcall  */
  int lgm;  /* local, global or member  */

  /* function recognition  */
  int fid;
  int sntip;

  /* fcall  */
  int fargsid;

  /* for structs */
  int ptip;
  int memb;
  int mbtip;

  /* dimensions for array   */
  int d1;
  int d2;

  /* location  */
  int row;
  int col;
};
```

Burada `char *tok` həddində tokenin mətndə olan sətir qarşılığı yerləşir. `unsigned long hid` -də isə həmin sətrin hash qarşılığı yerləşir. hash id –lər **`init_hash_ids();`** funksiyası tərəfindən mənimsədilir.

Tokenlər cərgəsi hazır olduqdan sonra parser işə salınır və parsinq ağacı qurulur.

parse.cpp

```
int parse(){

      init_parser();
      parser_core();

      print_tree_dx(tree);

      // Sleep(20000);
     //check_parse_errors();

      sehvler_yoxla();
```

```
    return 0;
}
```

Əvvəlcə tokenlər ardıcıllığı **init_parser()** funksiyası vastəsilə tok_list_el siyahısına köçürülür. Burada hər bir token-ə bir tok_list_el uyğun gəlir. tok_list_el –də ağacın budağı və ya yarpaqları yerləşir:

esrin.h

```
struct tok_list_el {
  struct token *tok; /* col and row staff  */
  int id; /* parsing */
  struct tok_list_el  *next;
  struct tok_list_el  *prev;
  struct tok_list_el  *childs[20]; /* max 20 child allowed, hope that's
enough  */
  int cld_cnt; /* number of childs of this childb   */
  int old_copy; /* indicates whether this node is copied from lower layer
                      directly , or by consyming transformed into new node
           possible bug! we set this filed to 1 on copy_to_upper_layer
          function, may that be set otherwise on layer constucting
functions?
    if so, then we lose control over the parse tree. For the moment I just
skip verification for late  */
  int cons_id; /* why consuming   */

  int lrb; /*  leave or branch: 0 - leave, 1 - branch
               all tokens initialized to leave  */
  int fargsid; /* id ind func args tab if it is fcall1   */
  /* exec band  */
  int head_pos;
  int band_id;
  int context_left;
  int sat[300]; //satisfaction
  int sat_len;
};
```

Daha sonra **parser_core()** funksiyası çağrılır. Biz ağacı yarpaqlardan kökə doğru tərsinə çevrilmiş formada qururuq. Bottom up parsing metodu ilə. **parser_core()** funksiyası bir neçə **bottom_up_parse_xxx** funksiyalarından ibarətdir.

```
int parser_core(){

  bottom_up_parse_dax_cap(tree);

  bottom_up_parse_exprs(tree);
```

```
bottom_up_parse_cnds(tree);

bottom_up_parse_seq(tree);

bottom_up_parse_fcall(tree);

. . .
```

Ağacın qurulması mərhələlər üzrə baş verir. Qurulma elə seçilmiş ardıcıllıq üzrə aparılır ki, bir qrammatik qaydanın tanınması digərinə mane olmasın, operatorların prioritrtlik dərəcəsi qorunsun. Misal üçün şərt operatorunu tanımaq üçün əvvəlcə şərti ifadələr tanınmalıdır: **(x+y*(3 - 5) >= 6) && (4 != x*z) .**

Şərti ifadələr **bottom_up_parse_cnds(tree);** funksiyası ilə tanınır. Şərti ifadələrin içərisində ədədi ifadələrdən istifadə olunduğuna görə şərti ifadələrdən də əvvəl ədədi ifadələr tanınmalıdır: **x+y*(3 - 5), x*z** v.s.

Ədədi ifadələr **bottom_up_parse_exprs(tree);** funksiyası ilə tanınır. Bu şəkildə ardıcıllığı gözləmək şərtilə addıdm addım daxil olan mətn proqramlaşdrıma dilinin qrammatik qaydalarına uyğun olaraq sözlərə, ifadələrə, cümlələrə və sonda yekun mətnə tərcümə olunur.

Hər bir **bottom_up_parse_xxx** funksiyası bir və ya bir neçə **build_cur_layer_xxx** funksiyasın çağırır.

```
void bottom_up_parse_cnds(struct tok_list *tree){

    while(build_cur_layer_cnds(tree));

}
```

Adətən bu **build_cur_layer_xxx** funksiyaları prosedur gərəyi **while** operatoru daxilində verilir. **build_cur_layer_xxx** funksiyaları tokenlər siyahısını bir dəfə soldan sağa oxumaqla tələb olunan qrammatik qaydanı ödəyən tokenlər ardıcıllığını müvafiq üst səviyyəli **budağa** çevirir. Əgər heç olmasa bir uyğun ardıcıllıq rast gəlindisə onda funsksiya 1 qaytarır və təkrar çağırılır (**while**).

Tutaq ki aşağıdakı tokenlər ardıcıllığı verilib.

```
5 + ( 6 * x) - 2 * ( 3 + y * (7 - z))
```

Birinci oxunuşda bu siyahı aşağıdakı kimi çevrilər.

```
5 + ( EXPR ) - 2 * ( 3 + y * (EXPR))
```

Bu oxunuşda yalnız `6 * x` və `7 - z` çevrildi. Çevrilmənin davamını başa düşmək üçün **build_cur_layer_xxx** funksiyası ilə tanış olaq. **build_cur_layer_xxx** funksiyaları bir və ya bir neçə **cons_xxx_tok** funksiyalarını çağırırlar – **consume** - udmaq. Yəni bi neçə tokeni **udaraq** bir token yarat.

```
int build_cur_layer_hp_oprs (struct tok_list *tree){

  struct tok_list_el *tll, *ptr;

  int  pos;

  tll = tree->first;

  pos = 0;      /* how many elements scanned so far  */


  while (tll!=NULL){

    ptr = create_tok_list_el(tll->tok); /* null reference */

    if (cons_crg1_tok(ptr, &tll,   &pos))
      continue;
    if (cons_crg2_tok(ptr, &tll,   &pos))
     continue;
    if (cons_brk_tok(ptr, &tll,   &pos))
```

Az öncə yuxarıda baxdığımız misalda `6`, `*` və `x` tokenləri EXPR tokeni ilə əvəz olundu. Bu əvəz olunma məhz **cons_xxx_tok** funksiyaları tərəfindən yerinə yetirilir.

```
int cons_mult_tok(struct tok_list_el *ptr, struct tok_list_el **tll_ref,
int *pos){

  struct tok_list_el *tllp = *tll_ref;

  struct tok_list_el *tll = tllp->next; /* because it is refeence */

  int a,b,c;

 if (tll!=NULL)
    a = tll->id;
  else
    return 0;

  if (tll->next!=NULL)
    b = tll->next->id;
  else
```

```
    return 0;

  if (tll->next->next!=NULL)
    c = tll->next->next->id;
  else
    return 0;


  if ((match_inaf(a) || matchid(a, EXPR) || matchid(a, BRK_VAL)) && \
        match_hp_opr(b) && \
       (match_inaf(c) || matchid(c, EXPR) || matchid(b, BRK_VAL) )){
          ptr->id  = EXPR;
       ptr->childs[0] = tll;
       ptr->childs[1] = tll->next;
       ptr->childs[2] = tll->next->next;
       ptr->cld_cnt   = 3;
       ptr->lrb = 1; //branch

       tllp->next = ptr;
       ptr->next = tll->next->next->next;
       *tll_ref = ptr;
    (*pos)++;
    return 1;
 }

  return 0;
}
```

Əgər birinci token dəyişən (İDT), ədəd (NUMB), cərgə (bir ölçülü CRG1, ikiölçülü CRG2), kəsr ədəd (FLOAT)

```
int match_inaf(int k){

  return matchid(k, IDT) || matchid(k, NUMB) || matchid(k, CRG1) ||
matchid(k, CRG2)  || matchid(k, FLOAT) ;

 }
```

və ya riyazi ifadə (EXPR) və ya mötərizə daxilində qiymətdirsə(BRK_VAL),

İkinci token isə yüksək prioritetli əməldirsə

```
int match_hp_opr(int k){

  return ( matchid(k, MULT) ||  matchid(k, DEL) ||  matchid(k, PRCT)) ;

}
```

üçüncü token də həmçinin Əgər birinci token dəyişən (İDT), ədəd (NUMB), cərgə (bir ölçülü CRG1, ikiölçülü CRG2), kəsr ədəd (FLOAT) , riyazi ifadə (EXPR) və ya mötərizə daxilində qiymətdirsə(BRK_VAL)

onda siyahıdan onları sil, yerinə yeni token EXPR yerləşdir.

```
ptr->id  = EXPR;
```

silinən tokenləri yenı yaradılan tokenlərin uşaq tokenləri kimi qeyd elə.

```
ptr->childs[0] = tll;
ptr->childs[1] = tll->next;
ptr->childs[2] = tll->next->next;
ptr->cld_cnt   = 3;
```

Bu formada bütün ifadələr daha üst səviyyəli qrammatik vahidlərə çevrilər: ifadələr və açar sözlər operatorlara, operatorlar bloklara, bloklar əvər varsa funksiyalara, funksiyalar isə yekun proqrama çevrilmiş olur. Yekun kod TEXT tokeni kimi işarə olunur. Əgər parsinq mərhələsinin sonunda sadəcə TEXT tokeni qalıbsa deməli tanınma uğurlu olub və mətndə heç bir səhv yoxdur, simvollar cədvəllərini yaratmaq prosesinə başlamaq olar.

Əks halda mətndə səhvlər var, onların yerini təyin edib və mümkün səbəblərini təxmin edib istifadəçiyə bildirmək lazımdır.

Yuxarıdakı ifadənin müvafiq parsinq ağacı aşağıdakı kimi olar:

```
FILEBEG EXPR MULT OPNBRK EXPR MULT BRK_VAL CLSBRK FILESON FILESON FILESON FILESON
FILESON
EXPR NEGSIG NUMB NUMB POSSIG IDT OPNBRK EXPR CLSBRK
NUMB POSSIG BRK_VAL NUMB NEGSIG IDT
OPNBRK EXPR CLSBRK
NUMB MULT IDT
```

Əvvəldəki FİLEBEG və sondakı 5 ardıcıl FİLESON tokenlərini biz özümüz siyahıya əlavə etmişik. Onlar heç bir qrammatik çevrilmədə iştirak eləmir, sadəcə siyahının sərhədlərini müəyyən etməyə kömək edir. Gördüyümüz kimi biz burda sonda TEXT tokeni almamışıq. Səbəb odur ki, daxil etdiyimiz mətn korrekt proqram mətni deyil. Mətni aşağıdakı kimi dəyişsək,

```
x = 5 + ( 6 * x) - 2 * ( 3 + y * (7 - z));
```

Parsinq ağacı bu şəklə düşər.

```
FILEBEG TEXT FILESON FILESON FILESON FILESON FILESON
SIMPLE_OPER
ASGN_OP
IDT ASGN EXPR NOQTEV
EXPR NEGSIG EXPR
NUMB POSSIG BRK_VAL NUMB MULT BRK_VAL
OPNBRK EXPR CLSBRK OPNBRK EXPR CLSBRK
NUMB MULT IDT EXPR MULT BRK_VAL
NUMB POSSIG IDT OPNBRK EXPR CLSBRK
NUMB NEGSIG IDT
```

Bu artıq korrekt proqram kodudur və ən yuxarıdakı TEXT tokeninə diqqət yetirək.

Simvollar cədvəlinin qurulması.

Kod generasiyası

İnterpretator

İnterpretator – İcraçı kompilyator tərəfindən gerenasiya olunan ikili faylı oxuyaraq ondakı instruksiyaları ardıcıl olaraq icra lentinə yığır. Daha sonra icraetmə göstəricisini ilk icaolunmalı instruksiyanın üzərinə kökləyərək proqramın icrasına başlayır.