Dokumentace k semestrální práci předmětu BI-PJP @ FIT ČVUT

Autor: Tomáš Kvasnička Rok: 2012/2013

Vlastní rozšíření

- *lexan.cpp/lexan.h* jsem rozšířil o klíčová slova pro zpracování cyklu for-to/downto a lexikání symboly "*[*", "*]*", "." potřebné k deklaraci pole. Dále jsem přidělal větev *q7* do funkce *readLexem()*, do které překladač vstupuje v případě parsování konstanty v osmičkové soustavě.

- *parser.cpp/parser.h* jsem rozšiřoval na několika místech.
  - V první řadě se jedná o vytvoření procedury *DeklPole(),* která zajišťuje syntaktickou kontrolu deklarace pole a volá funkci *deklPole()*, jenž provede samotnou alokaci adres pro prvky pole. Procedury *DeklProm()* a *ZbDeklProm()* bylo následně třeba upravit, aby možnost deklarace pole braly v potaz.
  - Funkci *Prikaz()* bylo třeba rozšířit o možnost práce s polem. Ve větvi *IDENT* je proto zjištěno, jestli identifikátor je identifikátorem pole a pokud ano, je provedena syntaktická kontrola pomocí procedury *Srovnani()* a jsou vytvořeny odpovídající uzly AST.
  - Ve funkci *Prikaz()* byla dále vytvořena nová větev *kwFOR* zajišťující vytvoření uzlů pro překlad for cyklů. Provádí syntaktickou kontrolu zápisu, vytváří uzly pro řídící proměnnou cyklu a přiřazení do ní a nakonec se dělí do dvou sub-větví přičemž jedna vytváří uzel pro překlad cyklu for-to a druhá uzel pro překlad cyklu for-downto.
  - Poslední upravovanou funkcí v těchto souborech je *Faktor(),* kde bylo opět třeba přidělat podporu pro pole. Ve větvi *IDENT* se tudíž také nachází kontrola datového typu identifikátoru a pokud je identifikátor identifikátorem pole je po syntaktické kontrole vytvořen odpovídající uzel AST.

- *strom.cpp/strom.h* byly taktéž upravovány. Došlo zde k vytvoření nových tříd reprezentujících uzly AST. Ve třídách byly implementovány metody pro optimalizaci a překlad těchto uzlů.
  - *class Arr:* reprezentuje jeden prvek pole. Obsahuje adresu prvku pole, ukazatel na výraz reprezentující index prvku a *bool* hodnotu vyjadřující zdali je prvek pole v příkazu přiřazení. Třída dědí od třídy *Expr*.
    - metoda *Optimize():* Optimalizuje hodnotu výrazu, který určuje index do pole.
    - metoda *Translate():* Provede překlad generováním instrukce *TA*, přeložením výrazu reprezentujícího index, generováním instrukce *BOP* a pokud uzel není použit v přiřazovacím příkazu tak i generováním instrukce *DR*. Instrukce *BOP* zde realizuje přičtení offsetu daného výrazem k adrese prvního prvku pole.
  - *class ArrAssign:* reprezentuje příkaz přiřazení výrazu prvku pole. Obsahuje ukazatel na uzel reprezentující prvek pole a ukazatel na uzel reprezentující výraz. Třída dědí od třídy *Statm.*
    - metoda *Optimize():* Volá metody *Optimize()* uzlu pole a uzlu výrazu.

        - metoda *Translate():* Volá metody *Translate()* uzlu pole a uzlu výrazu. Poté vygeneruje instrukci *ST*.
    - *class ForTo:* reprezentuje příkaz cyklu for-to. Obsahuje ukzatel na uzel řídící proměnné, ukzatel na uzel výrazu reprezentujícího horní limit cyklu, ukazatel na uzel přiřazení počatečního výrazu řídící proměnné a ukazatel na uzel reprezentující příkaz prováděný v těle cyklu. Třída dědí od třídy *Statm.*
        - metoda *Optimize()*: Optimalizuje příkazy prováděné v těle cyklu.
        - metoda *Translate():* Přeloží přiřazovací příkaz a následně uchová hodnotu čítače instrukcí. Poté přeloží uzel řídící proměnné, vygeneruje instrukci *DR* a přeloží uzel výrazu s horním limitem cyklu. Dále vygeneruje sled instrukcí k ověření, zdali je hodnota v řídící proměnné nižší nebo stejná než hodnota výrazu určujícího horní limit a následně vygeneruje instrukce těla cyklu. Nakonec generuje instrukce k ikrementaci hodnoty v řídící proměnné a instrukci *JU* jejímž parametrem je dříve uložený čítač instrukcí.
    - *class ForDownto:* implementačně i strukturálně stejná jako třída *ForTo* pouze místo inkrementace řídící proměnné probíhá její dekrementace a význam výrazu, který určoval horní limit běhu cyklu je interpretován jako spodní limit.

- *tabsym.cpp/tabsym.h* jsou soubory rozšířené o nové funkce a procedury a společně s tímto rozšířením zde proběhnulo několik úprav původního kódu. Ve všech funkcích byl upraven způsob práce se spojovým seznamem reprezentujícím tabulku symbolů a to tak, aby nové deklarace byly vkládány na konec tohoto seznamu, nikoliv na začátek jak tomu bylo v původní implementaci. Tato úprava byla zavedena z důvodu implementace pole. Index je totiž reprezentován jako offset, který se přičítá k adrese prvního prvku pole a bylo proto třeba, aby při vyhledávání v tabulce prvků byl jako první nalezen prvek na začátku pole.
    - *deklPole():* funkce ověří platnost identifikátoru a pakliže není stejný identifikátor již použitý alokuje adresy pro všechny prvky pole. Velikost pole je dána rozsahem mezi spodním indexem včetně tohoto indexu a horním idexem nevčetně. Spodní index musí být vždy roven nule.
    - *isArr():* voláním funkce *hledejID()* funkce ověří, zdali je identifikátor identifikátorem pole a pokud ano, vrací *true*. Pokud identifikátor není deklarován nebo se nejedná o identifikátor pole vrací funkce *false.*
    - *addrArr():* funkce ověří, že identifikátor je identifikátorem pole a pokud ano, vrátí adresu prvního prvku pole.

Ukázka kódu

- Implementace uzlu typu *Arr*

```
class Arr : public Expr
{
      int addr;
      Expr* idx;
      bool rvalue;
public:
      Arr(int, Expr*, bool);
      virtual Node* Optimize();
      virtual void Translate();
      virtual ~Arr();
};

Arr::Arr(int a, Expr* i, bool r)
{
      addr = a;
      idx = i;
      rvalue = r;
}

Arr::~Arr()
{
      delete idx;
}

Node* Arr::Optimize()
{
      idx = (Expr *)(idx->Optimize());
      return this;
}

void Arr::Translate()
{
      Gener(TA, addr);
      idx->Translate();
      Gener(BOP, Plus);
      if (rvalue)
            Gener(DR);
}
```