PROJEKT PŘEKLADAČE

pro předměty Formální jazyky a překladače a Algoritmy

Tým 097, varianta I

Tom Barbořík (xbarbo06) – 30%
Pavel Kaleta (xkalet05) – 30%
Miroslav Tichavský (xticha04) – 25%
David Myška (xmyska05) – 15%

Rozšíření: projekt neimplementuje žádné z možných rozšíření

Lexikální analyzátor

Pracoval: Pavel Kaleta

Výstup:

Token reprezentující jedem lexém.

Implementace:

Pomocí konečného deterministického automatu. Při zavolání funkce scanner začne číst znak po znaku ze vstupního souboru a podle jistých pravidel se rozhoduje mezi stavy a vrací příslušný token. Pokud se scanner dostane do chybného stavu, předá tuto skutečnost parseru, který to vrátí jako lexikální chybu.

Syntaktický analyzátor (bez zpracování výrazů)

Pracoval: Miroslav Tichavský, Tom Barbořík, Pavel Kaleta

Pro implementaci syntaktické analýzy jsme zvolili doporučenou metodu **rekurzivního sestupu**. Největším úskalím, před vlastní implementací, bylo správné zvolení pravidel pro konstrukci bezkontextové gramatiky, pomocí které jsme vytvořili LL-tabulku, tak aby odpovídala zadaným konstrukcím jazyka IFJ17.

Vstup:

Syntaktický analyzátor volá scanner (lexikální analyzátor), který v globální proměnné předkládá typ načteného tokenu a případně další potřebné detaily o tokenu.

```
95  /* BODY → scope EOL <PROGRAM> end scope */                  ----> Pravidlo pro rozgenerování neterminálu na levé straně
96  int func_body() {                                           ----> Funkce vykonávající rozgenerování daného neterminálu
97      int result;
98      if (token == M_SCOPE) {                                 ----> Kontrola symbolu na vstupu
99          token = scanner();                                  ----> Volání scanneru pro načtení dalšího tokenu
100         if (token == LEX_ERROR) return LEX_ERROR;           ----> V případě lexikální chyby ukončuji syntaktickou analýzu a vracím příslušný chybový kód
101         if (token == M_EOL) {
102             //...
103             result = func_program();                        ----> V případě dalšího netrminálu v pravidle, musím volat jemu příslušnou funkci
104             if (result != SYNTAX_OK) return result;         ----> V případě chyby propaguji danou chybu do volající funkce
105             //...
106             return SYNTAX_OK;                               ----> Pokud odsimulovaná derivace uspěla, propaguji syntaktickou správnost
107     }
108         return SYNTAX_ERROR;                                ----> V opačném případě propaguji syntaktickou chybu
109 }
```

Výstup:

Výstupem analýzy je vyhodnocení, zda je zdrojový kód zapsán syntakticky správně nebo došlo k syntaktické chybě. Dále dochází k naplnění tabulky symbolů a kontrole sémantiky. Také se zajišťuje vytvoření syntaktických stromů, které jsou následně předkládány generátoru cílového kódu.

Implementace:

Pro každý neterminál gramatiky je implementována funkce, která pomocí kontroly příchozích tokenů vyhodnocuje, zda tokeny korespondují s gramatickými pravidly pro daný neterminál (viz. LL-tabulka). Neterminály jsou zástupné symboly, které se během rekurzivního sestupu snažíme v několika krocích nahradit sekvencí terminálů.

Během rekurzivního volání funkcí dochází také ke kontrole sémantické správnosti zdrojového kódu, k tomu je využita tabulka symbolů implementována v modulu **symtable.c**. Do tabulky vkládáme identifikátory a další potřebné detaily o proměnných a funkcích a při jejich volání kontrolujeme, jestli došlo ke správné definici či deklaraci. Kontrolujeme také, jestli nedochází k jejich redefinici.

```
165  /*  F_DEC_DEF → declare function id (<PARAM_LIST>) as DATATYPE */
166  int func_f_dec_def() {                                  ---->  Funkce zpracovávající deklarace a definice funkcí
167  //...
168  if(token == M_ID) {                                     ---->  Pokud narazím při zpracovávání deklarace na identifikátor
169      char *id = str_to_array(detail);
170      s_btree node;
171      int resultTree = BTget(&g_symtable, id, &node);     ---->  Kontroluji jeho výskyt v globální tabulce symbolů
172
173      if (resultTree == 0) {
174          return SEMANTIC_ERROR_1;                        ---->  Pokud se jedná o redeklaraci, propaguji sémantickou chybu
175      //...
176          return SYNTAX_OK;                               ---->  Pokud odsimulovaná derivace uspěla, propaguji syntaktickou správnost
177      }
178      return SYNTAX_ERROR;                                ---->  V opačném případě propaguji syntaktickou chybu
179  }
```

Vykonáváním rekurzivního sestupu je zároveň prováděno plnění seznamu syntaktických stromů, které se následně předloží generátoru cílového kódu. Pro plnění jsou využity datové struktury a funkce implementovány v modulu **syntaxtree.c**.

```
484  /*  P_DEFINITION → dim id as DATATYPE <ASSIGN> */
485  int func_p_definition() {                               ---->  Funkce zpracovávající definici proměnné
486      //...
487      e_dtype type = d_void;
488      result = func_datatype(&type);                      ---->  Zjistím typ proměnné
489      //...
490      BTput(&t_symtable, id, sn_var, type, NULL);         ---->  Vložím proměnnou do lokální tabulky symbolů
491      Ladd(&list, STcreateVar(id, type));                 ---->  Přidám definici proměnné do seznamu syntaktických stromů pro generátor
492      //..
493  }
```

Precedenční analýza

Pracoval: Pavel Kaleta

Kontroluje správný zápis výrazu. Zda odpovídá pravidlům a možným typovým konverzím.

Vstup:

Seznam tokenů.

Výstup:

Hodnota, zda je výraz v pořádku, nebo je v něm syntaktická nebo sémantická chyba.

Implementace:

Pomocí zásobníku, který se řídí precedencí, což je priorita termů zaznamenaných v precedenční tabulce. Ta je realizovaná pomocí dvojdimenzionálního pole. Priorita načteného symbolu se porovná s prioritou nejvrchnějšího termu na zásobníku a podle toho se rozhodne, co se s daným tokenem provede. Pokud kombinace priorit těchto termů neexistuje, vrátí funkce chybu.

Postfix

Pracoval: Pavel Kaleta

Převod infixového výrazu na postfixový.

Vstup:

Hodnota indikující načtený symbol.

Výstup:

Pole reprezentující postfixový výraz.

Implementace:

Výraz se musí zbavit závorek a k tomu je využit zásobník, do kterého se podle algoritmu přidávají ty tokeny, které by měly být přeskočeny kvůli závorkám. Tyto tokeny jsou poté na příslušném místě za zásobníku vyndány a doplněny do výrazu.

Tabulka symbolů

Pracoval: David Myška

Tabulka symbolů je implementována pomocí binárního vyhledávacího stromu. Vyhledávání probíhá podle klíče odpovídajícímu názvu proměnné nebo funkce. Každý uzel si s sebou nese několik informací, typ uzlu, zda se jedná o funkci nebo proměnnou, datový typ, pro proměnné označuje její datový typ a pro funkce návratový typ. Dále pak seznam parametrů, který je definovaný pouze pro funkce.

Syntaktický strom

Pracoval: David Myška

Syntaktický strom jsme použili jako rozhraní mezi syntaktickou analýzou a generátorem výstupního kódu. Generátoru je pak předáván celý seznam stromů, implementovaný pomocí dynamického pole, obsahující vnitřní reprezentaci celého vstupního kódu.

Jednotlivé stromy, popřípadě jeho uzly, se vytvářejí pomocí skupiny funkcí "*`STcreate`*...", například funkce *`STcreateIntConst(int value)`* vytvoří uzel pro celočíselnou konstantu odpovídající hodnotě zadané jejím parametrem. Některé funkce mají jako parametr podstrom vytvořené také těmito funkcemi, proto jsou funkce "*`STcreate`*..." psány tak, aby je bylo možné zanořovat, například voláním "*`STcreateExpr(STcreateDoubleConst(1.75), "*", STcreateVar("var", d_double))`*" se vytvoří strom odpovídající výrazu "*1.75 * var*". Každá z funkcí přijímající nějaký podstrom jako parametr přísně kontroluje, zda zadaný parametr má povolený typ, například do podmínky nelze vložit výpis na obrazovku. Pro přidaní stromu do seznamu pak jen stačí zavolat funkci *`Ladd(s_list *list, s_stree tree)`*, popřípadě jako parametr "*tree*" lze opět zadat přímo příkazy ze skupiny "*`STcreate`*...".

Řetězec (String)

Pracoval: Tom Barbořík

Modul textového řetězce pro práci s proměnným počtem znaků.

Implementace

Řetězec je implementován pomocí lineárního acyklického seznamu. Seznam se skládá z uzlů, kde každý uzel obsahuje určitý kus výsledného řetězce. Podle konstanty *STR_MAXLEN* se rozhoduje, jak velký blok má být. V případě, že je v bloku nedostatek místa pro vkládaný řetězec, vytváří se nový, dokud není celý řetězec vložený. Je to z důvodu, aby se při přidání každého znaku neprováděla alokace paměti a tím se ušetřil čas, přestože výsledný *String* pak může zabírat více paměti, než by zabíral při alokaci po jednom znaku. Aktuálně je délka bloku nastavena na 64 znaků, protože ne příliš se objevovaly delší řetězce.

Zásobník

Pracoval: Tom Barbořík

Modul obecného zásobníku.

Implementace

Zásobník je implantován pomocí dvou struktur. Hlavní obsahuje informace o vrcholu a druhá už reprezentuje samostatný prvek s daty a odkazem na další prvek. Použité funkce byly inspirované látkou z předmětu IAL.

Generátor

Pracoval: Tom Barbořík

Generátor slouží pro tvorbu výsledného kódu ze seznamu syntaktických stromů. Výsledný kód odpovídá pravidlům interpretu a reprezentuje vstupní kód jazyka IFJ17.

Implementace

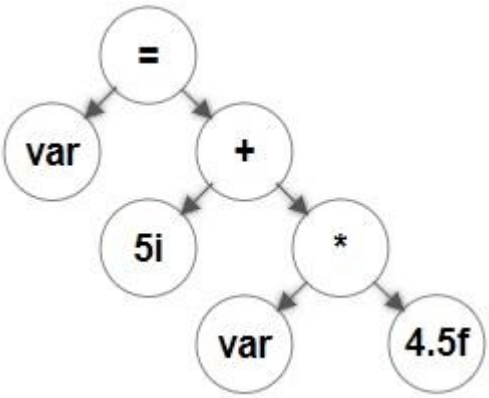
Poté, co dokončí parser svoji práci, je zavolán generátor. Ten jako první vytvoří hlavičku výsledného kódu společně s kódem pro vytvoření lokálního rámce a poté projde celý seznam syntaktických stromů a pro každou položku zavolá určité. To se rozliší podle typu položky, např. *n_var*, *n_expr* … . Nejzajímavější je implementace podmínek, smyček a výrazů.

Podmínky a smyčky (if, while) jsou implementované pomocí zásobníku, který se využívá pro podporu zanořování řídících struktur. Při tvorbě každé struktury je každé přiděleno unikátní id, jež je využito pro tvorbu návěští. Toto id a uzel popisující strukturu jsou vloženy na vrchol zásobníku, ze kterého se při ukončení řídící struktury odebírá.

Podmínkové výrazy jsou v alternativě zpracované pomocí funkce, která generuje výrazy navíc s pomocnou proměnnou, ve které je uložena pravdivostní hodnota. Podle toho se poté rozhodne o přeskočení bloku s podmínkou nebo její vykonání. Pomocné proměnné se v téměř všech situacích ukládají na lokální rámec.

Cykly byly na implementaci komplikovanější, protože v nich definované pomocné proměnné každý cyklus znovu definují. To bylo vyřešeno nastavením flagu, že se momentálně generuje kód uvnitř cyklu. Při tomto nastavení jsou všechny pomocné proměnné definované na dočasný rámec, který se

vytváří nový pro každý cyklus. Tímto bylo zabráněno předefinování proměnné na lokálním rámci, což by vedlo k chybě interpretu. Jinak se cyklus chová stejně jako podmínka.



Nejsložitější ze všeho bylo vytvořit funkci pro vytvoření výrazu. Ty jsou ve stromě uspořádané stylem viz. obrázek, který reprezentuje přiřazení: `var = 5 + var * 4.5`.
Zde bylo obtížné vymyslet, jak strom převést, v jakém okamžiku vytvořit pomocné proměnné pro dočasné uložení výsledku a dále konverzi hodnot při operacích, které ji povolují. To se jedná pro typy float -> int a int <- float.
Celý algoritmus pracuje rekurzivně s průchodem postupně od nejpravějšího.
V případě, který je uveden na levém obrázku, se první zanoří funkce do nejpravějšího uzlu úplně na dně stromu. V něm se jedná o konstantu, tak se vytvoří pomocná proměnná, kterou funkce vrátí (reprezentuje konstantu). Pak se jde do levého uzlu, ve kterém je proměnná, a protože je konstanta typu float, vytvoří se podmíněné přetypování (pokud není var float, přetypuj na něj) a provede se vynásobení, jehož výsledek je opět uložen v pomocné proměnné, která je vrácena jako pravý operand pro sčítání. Tam se jde doleva, vytvoří se pomocná pro integer 5 a ten se přetypuje na float. Vrátí se a obě pomocné se sečtou. Výsledek se opět vrátí v pomocné, která se přesune do proměnné var. Pokud je proměnná typu int, po přiřazení se provede přetypování, tj. zaokrouhlení na celé číslo s prioritou sudé číslice.

Rozdělení bodů

Rozdělení bodů je nerovnoměrné, protože všichni členové týmu nevěnovali projektu stejné úsilí. Dalo by se říct, že Mirek splnil svou část zadání, nicméně v ní byla spousta chyb, které Tom a Pavel museli zdlouhavě opravovat a při finálním debuggingu Mirkovu část i docela dost rozšiřovat. David dostal za úkol jednoduchou část, a proto měl týmu pomoct, kdyby se vyskytl problém. Jenže když tým něco potřeboval, nikdy neměl čas, a tak neudělal nic dalšího. Tom a Pavel převzali nedokončenou práci po všech ostatních a dali ji dohromady.