Inst. f. Praktische Informatik — Übungen zu Übersetzerbau WS 2003/2004

Prof. Dr. H. Mössenböck — **Übung 4**

Zuname _____ Matr. Nr. _____

Übungsgruppe: Punkte _____ korr. _____

- ❑ 1 (Wöß) Do 10^{15} - 11^{45}
- ❑ 2 (Wöß) Do 12^{00} - 13^{30} Letzter Abgabetermin:
- ❑ 3 (Rammerstorfer) Do 17^{15} - 18^{45} Donnerstag, 27.11.2003, 8^{15} Uhr

---

1. Symbolliste (14 Punkte)

Erweitern Sie Ihren Parser um eine Symbolliste (Klasse *Tab*). Die dafür notwendigen Klassen befinden sich in einem eigenen Package *ssw.mj.symtab* (Angabe ohne access modifier) und haben die folgenden Schnittstellen:

```
class Obj {
  static final int Con, Var, Type, Meth, Fld, Prog;

  int kind;               // Con, Var, Typ, Fld, Meth, Prog
  String name;
  Struct type;
  Obj next;               // next local object in this scope
  int adr;                // Con: value; Meth, Var, Fld: offset
  int level;              // Var: declaration level; Meth: # of parameters
  Obj locals;             // Meth: reference to list of local variables
                          // Prog: symbol table of program
}

class Struct {
  static final int None, Int, Char, Arr, Class;

  int kind;               // None, Int , Char, Class, Arr
  Struct elemType;        // Arr: type of array elements
  int n;                  // Class: # of fields
  Obj fields;             // Class: reference to list of fields
}

class Scope {
  Scope outer;            // reference to enclosing scope
  Obj locals;             // symbol table of this scope
  int nVars;              // # of  variables in this scope
}

class Tab {
  static final Struct noType, intType, charType, nullType;  // predefined types
  static final Obj noObj;                                   // predefined objects
  static Obj chrObj, ordObj, lenObj;                        // predefined objects

  static Scope topScope;  // current scope
  static int level;       // nesting level of current scope

  static void init ();
  static void openScope ();
  static void closeScope ();
  static Obj insert (int kind, String name, Struct type);
  static Obj find (String name);
  static Obj findField (String name, Struct type);
}
```

Die Methode *init* initialisiert die Symbolliste und trägt alle vordeklarierten Namen (Funktionen, Typen und Konstanten) von MicroJava ein.

Die Methoden *openScope* und *closeScope* legen einen neuen *topScope* an bzw. entfernen den aktuellen *topScope* und erhöhen bzw. vermindern den aktuellen *level*.

Die Methode *insert* erzeugt ein Symbollistenobjekt (Klasse *Obj*), trägt seine Attribute ein und fügt es im aktuellen Gültigkeitsbereich in die Symbolliste ein. Wenn dort bereits ein Eintrag mit dem gleichen Namen vorhanden ist, soll ein semantischer Fehler ausgegeben werden.

Die Methoden *find* und *findField* dienen dazu, später auf die Symbollisteneinträge zugreifen zu können. *findField* wird für die Erzeugung der Symbolliste noch nicht benötigt.

find sucht nach einem Namen beginnend im aktuellen bis zum äußersten Gültigkeitsbereich.

findField sucht nach einem Namen in einer inneren Klasse, deren *Struct* in der Schnittstelle mitgegeben wird.

2. Fehlerbehandlung (10 Punkte)

Erweitern Sie Ihren Parser derart, dass er die Analyse nicht mehr beim ersten erkannten Fehler abbricht, sondern nach der Methode der *speziellen Fangsymbole* fortsetzt. Fügen Sie dazu zwei Synchronisationspunkte in Ihre Implementierung ein:

1. Wenn bei einer Reihe von aufeinanderfolgenden Deklarationen (*ConstDecl*, *VarDecl*, *ClassDecl*) ein Fehler auftritt, so soll unmittelbar nach der fehlerhaften Deklaration wieder aufgesetzt werden. Beschränken Sie sich dabei nur auf globale Deklarationen, d.h. Sie brauchen Variablendeklarationen innerhalb von Klassen oder Methoden nicht berücksichtigen.
2. Wenn bei einer Reihe von aufeinanderfolgenden Statements ein Fehler auftritt, so soll beim nächsten Statement (nach dem fehlerhaften) wieder aufgesetzt werden.

Suchen Sie also in der MicroJava-Grammatik jene Stellen, an denen diese Synchronisationspunkte eingefügt werden müssen und implementieren Sie für den Wiederaufsatz die Methoden *recoverDecl* und *recoverStat*, die jeweils die Analyse nach einem Fehler in einer Deklaration oder einem Statement fortsetzen.

Versuchen Sie dabei, irreführende Folgefehlermeldungen zu unterdrücken. Bedenken Sie auch, dass sich *ident* nicht ideal als Fangsymbol eignet und verwenden Sie ev. zusätzlich semantische Informationen, um bei bestimmten Namen doch wiederaufzusetzten (z.B. wenn es sich um Typbezeichnungen handelt).

Da Ihr Parser während eines Analysevorgangs nun mehrere Fehler entdecken kann, ist es sinnvoll, eine eigene (innere) Klasse *Errors* für die Fehlerstatistik einzuführen. *Errors* dient zum Zählen der aufgetretenen Fehler und auch zur Ausgabe von Fehlermeldungen. Bedenken Sie, dass Sie nun auch Semantikfehler erkennen können, die Sie durch die Methode *semError* von den Syntaxfehlern (Methode *synError*) unterscheiden können.

Implementieren Sie diese Klasse und verwenden Sie sie anstatt der bisherigen Fehlerausgabe. Ihre Parser-Klasse soll also um folgende Elemente erweitert werden:

```
static void recoverStat () {...}
static void recoverDecl () {...}

static class Errors {
  static void synError (String msg) {...}
  static void semError (String msg) {...}
  static int count () {...}
```