Übersetzerbau Cluj-Napoca, Rumänien 2007

Markus Löberbauer

Institut für Systemsoftware
Johannes Kepler Universität, 4040 Linz, Österreich

Loeberbauer@ssw.jku.at

2007-02-01

Zusammenfassung

In dieser Übung soll eine vollständiger Übersetzer für *Micro*, eine sehr einfach Sprache, entwickelt werden. Als Zielplattform dient die .Net-Runtime.

Projekt

Für das Projekt werden Templatedateien angegeben, um eine einheitliche Schnittstelle zu schaffen. Die Templagtedateien können von der Übungsseite heruntergeladen werden. In den Dateien befinden sich TODO Kommentare um die Stellen zu markieren an denen gearbeitet werden soll.

1 Scanner – Lexikalische Analyse

Der erste Teil unseres Übersetzers ist die lexikalische Analyse. Die Sprache ist einfach gehalten und bietet die in Listing 2 gezeigten Terminalsymbole. In den Terminalen werden die Zeichenklassen aus Listing 1 verwedet.

Listing 1: Zeichenklassen

```
char=beliebiges Zeichen.
digit="0..9".
idChar="a..z" + "A..Z" + "_".
```

Listing 2: Terminale

```
ident      = idChar { idChar | digit }.
charLit    = "'" char "'".
numberLit = digit { digit }.

ifKW       = "if".     elseKW     = "else".
endKW      = "end".    callKW     = "call".
funtionKW = "function".
plus       = "+".      minus      = "-".
eql        = "=".      neq        = "#".
lss        = "<".      gtr        = ">".
comma      = ",".      methPar    = "<-".
```

Die Scanner-Schnittstelle enthält eine Methode, *IToken Next()*. Der Aufruf der Methode Next() liefert das nächste Terminalsymbol (Token). Ist das Ende der Eingabedatei erreicht, soll bei jedem Aufruf ein *end of file* Token geliefert werden. Dabei müssen Whitespaces wie Leerzeichen, Zeilenumbrueche und Tabulatoren überlesen werden. Kommentare sind in Micro in geschweifte Klammern eingeschlossen, '{' beginnt einen Kommentar, '}' beendet einen Kommentar. Kommentare koennen auch geschachtelt auftreten. Beispiel: '{ Das ist ein {Geschachtelter } Kommentar}.'

1.1 Fehlerbehandlung

Der Scanner muss mit fehlerhaften Eingaben zurechtkommen, die Fehler aus der folgenden Liste müssen behandelt werden.

- "Invalid Character Literal"
- "Unexpected input character: " + ch
- "Number Overflow"
- "End of File in Comment"

2 Parser – Syntaktische Analyse

Der Parser ist als Top-Down-Parser im rekursiven Abstieg zu implementieren. Implementieren Sie für *jede* Produktion der Grammatik eine Parsmethode. Die Grammatik ist in Listing 3 abgebildet.

Listing 3: Terminale

```
Micro = { Function }.
Function =
        "function" ident ident [ "<-" FormPar { "," FormPar } ]
```

```
        Block
        "end"
.
FormPar = ident ident.
Block = { Statement }.
Statement =
        ( ident
                ( ident
                | "=" Expr
                | "<-" (charLit | numberLit )
                )
        | "if" Cond Block [ "else" Block ] "end"
        | Call
        )
.
Expr = Term { ("+" | "-") Term }.
Term = ( charLit | numberLit | ident | Call ).
Cond = Expr ( "<" | "=" | "#" | ">" ) Expr.
Call = "call" ident [ "<-" Expr { "," Expr } "end" ].
```

2.1 Fehlerbehandlung

Im Parser müssen die folgenden Fehler behandelt werden. Achtung, manche Fehler können erst später, unter Zuhilfenahme der Symboltabelle gemeldet werden.

- "Literal expected"
- "Incompatible types"
- "Illegal start of statement"
- "Parameter type mismatch"
- "Too many actual parameters"
- "Too few actual parameters"
- "Invalid conditional operator"
- "Invalid Term"
- "unexpected token, expected: " + expected token kind + ", found: " + found token kind"

3 Symboltabelle

Die Symboltabelle verwaltet bennante Symbole und Scopes (Gültigkeitsbereiche). In die Symboltabelle müssen alle benannten Elemente

eingetragen werden, Funktionen, Variablen und Typen. In Micro ist wird ein Scope durch eine Funktion oder einen if-Block geöffnet.

3.1 Fehlerbehandlung

In Micro ist es innerhalb eines Scopes nicht erlaubt Variablen mit dem gleichen Namen zu deklarieren. Funktionen und Variablen müssen vor ihrer Verwendung deklariert werden. Folgende Fehler können in diesem Zusammenhang auftreten.

- name + " declared twice"
- "Type not found: ' " + name + " ' "
- "Function not found: ' " + name + " ' "
- "Variable not found: ' " + name + " ' "

4 Codeerzeugung

Die Codeerzeugung wird in unserem Fall benutzt, um Micro Quellcode in ein .Net 2.0 Assembly zu übersetzen. Als Beispiel sei eine einfache if-Anweisung gegeben, siehe Listing 4. Im Beispiel wird links der Micro-Quellcode und rechts der ILAsm-Code angegeben.

Listing 4: if

```
int x
x = 1
                        1: ldc.i4.1
                        2: stloc.0
if 1 < 39
                        3: ldc.i4.1
                        4: ldc.i4 39
                        5: blt IL_7
                        6: br IL_11
  x = x + 1
                        7: ldloc.0
                        8: ldc.i4.1
                        9: add
                       10: stloc.0
end
put_i <- x end
                       11: ldloc.0
                       12: call int32 Calc::put_i(int32)
```

4.1 Fehlerbehandlung

Bei der Codeerzeugung können folgende Fehlermeldungen auftreten.

- "Unexpected symbol kind: " + sym.Kind
- "Compiler error in Code.load, unexpected item kind"
- "Left-hand side of an assignment must be a variable"
- "Cannot create code item for this kind of symbol table object"
- "Missing 'main' function"