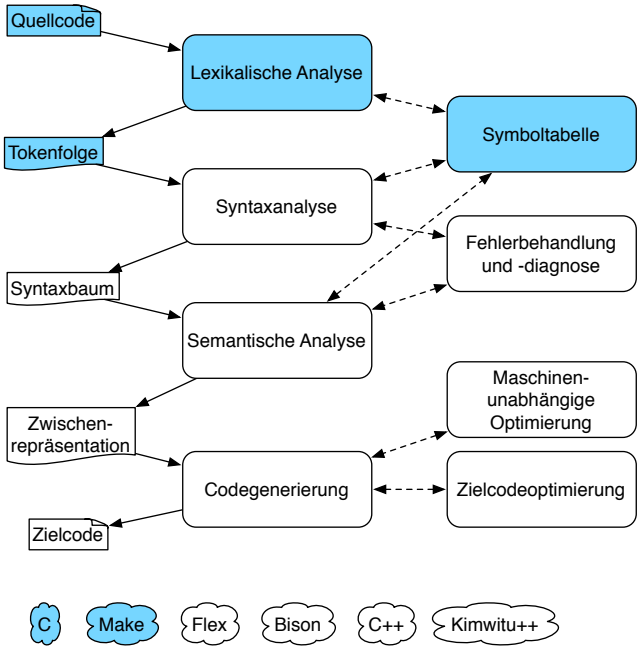
Rückblick

In den ersten Übungen wurden Sie mit der Speicherverwaltung und der Behandlung von Ein- und Ausgaben in der Programmiersprache C konfrontiert. Im Rahmen der Vorlesung Compilerbau können Sie nun rudimentär eine Eingabe on-the-fly in Tokenfolgen aufteilen.

Ausblick

In der zweiten Serie soll der Umgang mit der Sprache C sowie dem Werkzeug Make vertieft werden und ein etwas größeres Programm aus mehreren Komponenten aufgebaut werden. Inhaltlich steht die lexikalische Analyse von Modula 2 und die Verwaltung einer Symboltabelle im Mittelpunkt.



Referenzen

- Modula-2-Sprachreferenz (siehe StudIP)
- C Reference, http://en.cppreference.com/w/c
- Man-Pages: fopen, fclose, fprintf, fgetc, sprintf, fscanf, fputc, strcmp, strtok, strcpy, strncpy, malloc, calloc, realloc, memset, free, isalpha, isdigit, exit
- Headers: stdlib.h, stdio.h, ctype.h, string.h
- GNU Make Manual, http://www.gnu.org/software/make/manual

Lexikalische Analyse

Schreiben Sie ein C-Programm mit dem Namen `moco`, welches in der Lage ist, eine Textdatei in ihre Token zu zerlegen. Diese Token sollen dann wie folgt ausgegeben werden:

- Token ohne Wert (Schlüsselworte, Operatoren, Trennzeichen): Name des Tokens in spitzen Klammern, z.B. `<KEY_MODULE>` oder `<SEMICOLON>`.
- Token mit Wert (Konstanten): Name des Tokens, gefolgt von einem Komma, einem Leerzeichen und dem Wert – dies alles in spitzen Klammern, z.B. `<IDENT, HelloWorld>` oder `<INTEGER, 2>`. Bei Zeichen und Zeichenketten sollen die Anführungszeichen nicht mit gespeichert oder angezeigt werden: `<CHAR, c>` und `<STRING, Dies ist ein Test>`.
- Kommentare und Whitespace sollen grundsätzlich ignoriert werden. Bei der Ausgabe sollen die Tokens durch Leerzeichen getrennt sein. Sie müssen sich nicht um Zeilenumbrüche in der Ausgabe kümmern.

Parameter

Das Programm soll auf verschiedene Weisen (wie in Serie 1) aufrufbar sein:

- `moco`

  Aufruf ohne Parameter: lese von der Standardeingabe und schreibe auf die Standardausgabe

- `moco infile`

  Aufruf mit einem Parameter: lese von der Datei `infile` (sofern diese lesbar ist) und schreibe auf die Standardausgabe

- `moco infile outfile`

  Aufruf mit einem Parameter: lese von der Datei `infile` (sofern diese lesbar ist) und schreibe in die Datei `outfile` (sofern diese schreibbar ist)

Beispiel

Für eine Eingabedatei `hello.m` mit dem Inhalt

```
1  MODULE HelloWorld;
2
3  TYPE
4  myInt = INTEGER;
5
6  CONST
7  pi = 3.1415926;
8
9  VAR
10 x : REAL;
11
12 BEGIN
13 f := 2 + pi;
14 END HelloWorld.
```

würde das Programm wie folgt arbeiten:

```
$ ./moco hello.m
<KEY_MODULE> <IDENT, HelloWorld> <SEMICOLON> <KEY_TYPE> <IDENT, myInt> <EQ> <KEY_INTEGER>
<SEMICOLON> <KEY_CONST> <IDENT, pi> <EQ> <REAL, 3.141593> <SEMICOLON> <KEY_VAR> <IDENT, x>
<COLON> <KEY_REAL> <SEMICOLON> <KEY_BEGIN> <IDENT, f> <ASSIGN> <INTEGER, 2> <PLUS> <IDENT, pi>
<SEMICOLON> <KEY_END> <IDENT, HelloWorld> <PERIOD>

$ ./moco hello.m output

$ cat output
<KEY_MODULE> <IDENT, HelloWorld> <SEMICOLON> <KEY_TYPE> <IDENT, myInt> <EQ> <KEY_INTEGER>
<SEMICOLON> <KEY_CONST> <IDENT, pi> <EQ> <REAL, 3.141593> <SEMICOLON> <KEY_VAR> <IDENT, x>
<COLON> <KEY_REAL> <SEMICOLON> <KEY_BEGIN> <IDENT, f> <ASSIGN> <INTEGER, 2> <PLUS> <IDENT, pi>
<SEMICOLON> <KEY_END> <IDENT, HelloWorld> <PERIOD>
```

Kommentare sollen ignoriert werden.

```
$ echo "(* Dies ist ein Kommentar *)" | ./moco
```

Gehen Sie zur Vereinfachung davon aus, dass Kommentare nicht geschachtelt sind.

Fehlermeldungen

Falls es ein Problem mit den Eingabe- oder Ausgabedateien gibt oder der Nutzer zu viele Parameter angibt, soll das Programm (wie in Serie 1) mit einer Fehlermeldung und dem Exit-Code 1 abbrechen.

```
$ ./moco unlesbaredatei.m
error: cannot open file for reading

$ ./moco hello.m nichtschreibbaredatei.m
error: cannot open file for writing

$ ./moco hello.m output ueberfluessigerparameter
error: wrong number of parameters
```

Weiterhin gibt es in dieser Serie erstmals die Situation, dass `moco` mit unerwarteten Zeichen konfrontiert wird. In solchen Fällen soll das Programm ebenfalls mit einer Fehlermeldung und dem Exit-Code 1 abbrechen.

```
$ echo "?" | ./moco
error: unexpected character
last read character: '?'

$ echo "'bar'" | ./moco
error: unexpected character
last read character: 'a'

$ echo "(* Kommentar, der nicht endet" | ./moco
error: unexpected end of file
```

Die Ausgabe bei Fehler wird über die vorgegebene Funktion `void yyerror(char const*)` realisiert.

Makefile

In dieser Serie werden viele Quelltexte vorgegeben, weswegen ein Makefile genutzt wird, um die Übersetzung so modular wie möglich zu halten. Beim Aufruf von `make` sollen zunächst alle Quelldateien (z.B. `main.c`) in Objektdateien (z.B. `main.o`) übersetzt und diese anschließend zum ausführbaren Programm (`moco`) gelinkt werden. Der Aufruf von `make clean` löscht alle generierten Dateien.

Vorgaben

- Mittels `make` muss sich das lauffähige Programm `moco` erstellen lassen.
- Beschreibung der einzelnen vorgegebenen Dateien:
  - `buffer.*` – Diese Dateien sind vorgegeben und implementieren einen Zeichenpuffer. Es ist stets möglich, über die globalen Variablen `char* yytext` und `int yylen` auf den Wert und die Länge des aktuell gelesenen Tokens zuzugreifen. Der Puffer muss initial mit `void buffer_init()` initialisiert werden und am Programmende mit `void buffer_destroy()` zerstört werden. Diese Dateien sollen nicht verändert werden.
  - `diagnosis.*` – Diese vorgegebenen Dateien stellen die Funktion `void yyerror(char const*)` zur Verfügung, mit der Fehler angezeigt werden und das Programm abgebrochen wird. Diese Dateien sollen nicht verändert werden.
  - `scanner.*` – In dieser Datei wird mit der Funktion `char next_char()` ein Zeichen von `yyin` gelesen. Weiterhin werden in dieser Datei einzelne Funktionen zum Scannen von Konstanten, also IDENT , INTEGER , REAL , CHAR und STRING implementiert. Die Datei `scanner.h` soll nicht verändert werden – sämtliche Implementierungen finden in der Datei `scanner.c` statt.
  - `tokens.h` – In dieser Datei wird eine Tokenliste verwaltet Mit der Funktion `void append(token_t)` können Tokens hinzugefügt werden. Über die Funktion `void output_token_list()` erfolgt die Ausgabe auf `yyout`. In der Datei `tokens.h` ist lediglich für die Zusatzaufgabe eine Änderung notwendig und kann ansonsten beibehalten werden. Alle weiteren Implementierungen finden in der Datei `tokens.c` statt.
  - `symboltable.*` – Diese Dateien werden in der Zusatzaufgabe benötigt und beschrieben.
- Das eigentliche Programm soll in der Datei `main.c` implementiert werden, wo zunächst die Kommandozeilenparameter ausgewertet werden sollen, dann die Ein- und Ausgabe initialisiert wird, die Eingabe gescannt und die Tokenliste ausgegeben wird. Vergessen Sie nicht, allokierten Speicher freizugeben und die Dateien zu schließen.

Verwaltung von Bezeichnern

In einer Symboltabelle sollen Bezeichner verwaltet werden und am Ende des Programmes zu jedem Bezeichner die Zeile angegeben werden, in der er definiert wurde.

Für das Beispiel oben wäre die Ausgabe also:

```
$ ./moco hello.m
<KEY_MODULE> <IDENT, HelloWorld> <SEMICOLON> <KEY_TYPE> <IDENT, myInt> <EQ> <KEY_INTEGER>
<SEMICOLON> <KEY_CONST> <IDENT, pi> <EQ> <REAL, 3.141593> <SEMICOLON> <KEY_VAR> <IDENT, x>
<COLON> <KEY_REAL> <SEMICOLON> <KEY_BEGIN> <IDENT, f> <ASSIGN> <INTEGER, 2> <PLUS>
<IDENT, pi> <SEMICOLON> <KEY_END> <IDENT, HelloWorld> <PERIOD>
HelloWorld - defined in line 1
f - defined in line 13
myInt - defined in line 4
pi - defined in line 7
x - defined in line 10
```

Die Bezeichner sollen alphabetisch sortiert in genau dem gezeigten Format ausgegeben werden. Implementieren Sie die Symboltabelle in den Dateien `symboltable.c` und `symboltable.h`. Gewisse Vorgaben hierzu finden Sie bereits in der Datei `symboltable.h`. Sie können in der Tokenliste auch Einträge in der Symboltabelle referenzieren. Dazu müssten Sie einsprechende Änderungen in `tokens.*` vornehmen.

Hinweis

- Auch innerhalb von Kommentaren kann es Zeilenumbrüche geben.