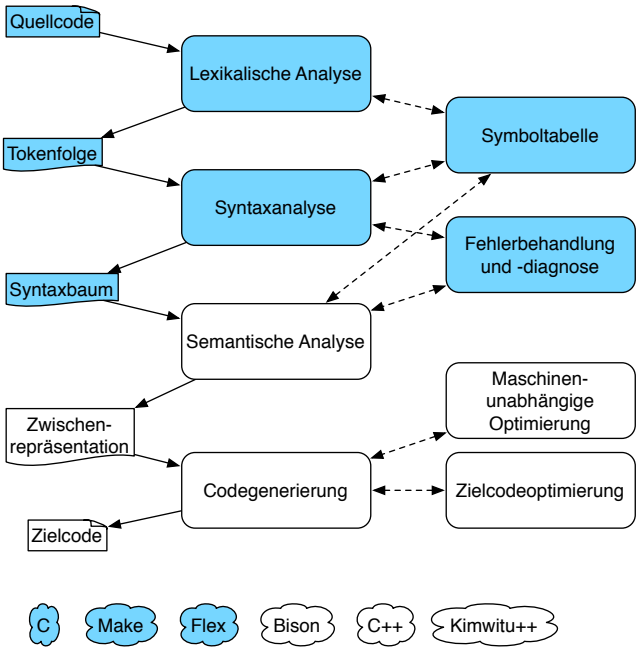
Rückblick

In Serie 2 mussten Sie mit Makefiles und fremdem Code arbeiten. Gerade letzteres ist ein wichtiger Aspekt, um das so genannte *Not-invented-here-Syndrom* zu vermeiden. Inhaltlich haben Sie reguläre Ausdrücke in endliche Automaten übersetzt und einen Scanner gebaut, der auch Kommentare verarbeitet. Zusammen mit einer Symboltabelle haben Sie damit die lexikalische Analyse grundsätzlich abgeschlossen.

Ausblick

In der dritten Serie soll das erste Metawerkzeug, Flex, genutzt werden und der generierte Code in das zu entwickelnde Werkzeug integriert werden. Inhaltlich steht neben der lexikalische Analyse nun die syntaktische Analyse von Modula-2 im Vordergrund.



Referenzen

- Modula-2-Sprachreferenz - StudIP: Modula-2 Sprachreferenz
- C Referenz - http://en.cppreference.com/w/c
- GNU Make Manual - StudIP und http://www.gnu.org/software/make/manual
- Flex Manual - StudIP und http://flex.sourceforge.net/manual
- Doxygen, http://www.stack.nl/~dimitri/doxygen/

Aufgabe 1: Top-Down-Parser

Schreiben Sie ein C-Programm mit dem Namen `moco`, welches nach dem Prinzip des rekursiven Abstieges (LL(1)-Verfahren) die syntaktische Analyse für Modula-2-Programme realisiert. Die lexikalische Analyse soll dabei durch ein vom Metawerkzeug Flex generierten Scanner übernommen werden.

Für diese Aufgabe sind zwei Teilaufgaben zu lösen:

1. **Spezifizieren der Lexik im Flex-Format in der Datei `scanner.l`.**
   Arbeiten Sie dabei die Lexik in der Sprachreferenz vollständig durch und spezifizieren Sie die regulären Ausdrücke für Schlüsselworte, Operatoren, Trennzeichen, Kommentare, Whitespace und Konstanten. Aus den regulären Ausdrücken generiert Flex eine Funktion `int yylex()` in der Datei `scanner.c`. Um mit dem restlichen Programm zu interagieren, nutzen Sie die in der Datei `parser.h` definierten Konstanten des Typs `yytokentype`. Damit ist es möglich, Regeln der Form

   ```
   "ELSIF" { return KEY_ELSIF; }
   ```

   zu spezifizieren. Das Makro `KEY_ELSIF` ist dabei vom Typ `yytokentype`. Mit dem in `parser.c` implementierten Arrays `char* tokenNames[]` kann wiederum der Name des Tokens angezeigt werden.
2. **Implementieren des Top-Down-Parsers in der Datei `parser.c`.**
   Folgen Sie hier dem in der Vorlesung vorgestellten Prinzip. Dies bedeutet, dass für jedes Nichtterminalzeichen der Grammatik eine eigene Funktion in C programmiert werden soll, bspw. die Funktion `void module()` für das Nichtterminal `module`. Diese Funktionen überprüft jeweils die Existenz der notwendigen Terminalzeichen (in Zusammenarbeit mit dem generierten Lexer) und ruft entsprechend der Grammatik und der Eingabe im rekursiven Abstieg gegebenenfalls weitere Funktionen für Nichtterminalzeichen auf.
   Nutzen Sie zur Verarbeitung der Nichtterminalzeichen die Funktionen `void NEXT()` zum Lesen des nächsten Tokens, `int TEST(yytokentype s)` zur Überprüfung, ob das aktuell gelesene Token mit einem übergebenem `s` übereinstimmt und `void MATCH(yytokentype s)` um die Eingabe gegen ein übergebenes Token `s` zu matchen.
   Um die Arbeit des Top-Down-Parsers nachvollziehen zu können, sollen Ausgaben bei jedem Betreten und Verlassen von Funktionen für Nichtterminalzeichen, sowie beim Lesen von Terminalzeichen aus der Eingabe gematcht werden (siehe Beispiel).

Es sind eine Reihe von Dateien und Funktionen vorgegeben. Diese sind im Punkt *Vorgaben* beschrieben.

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

```
(*
Unser beliebtes Hello World-Programm
*)

MODULE HelloWorld;

TYPE
myInt = INTEGER;

CONST
pi = 3.14159265358979323846264338327950288419716939937510582097494 4;

VAR
x : REAL;

BEGIN
x := 2 + pi;
END HelloWorld.
```

würde das Programm wie folgt arbeiten:

```
$ ./moco hello.m
read symbol in line 5: KEY_MODULE
entering module
|  read symbol in line 5: IDENT (HelloWorld)
|  read symbol in line 5: SEMICOLON
|  read symbol in line 7: KEY_TYPE
|  entering block
|  |  entering declaration
|  |  |  read symbol in line 8: IDENT (myInt)
|  |  |  entering type_declaration
|  |  |  |  read symbol in line 8: EQ
|  |  |  |  read symbol in line 8: KEY_INTEGER
|  |  |  |  entering type_denoter
|  |  |  |  |  read symbol in line 8: SEMICOLON
|  |  |  |  leaving type_denoter
|  |  |  leaving type_declaration
|  |  |  read symbol in line 10: KEY_CONST
|  |  leaving declaration
|  |  entering declaration
|  |  |  read symbol in line 11: IDENT (pi)
|  |  |  entering constant_declaration
|  |  |  |  read symbol in line 11: EQ
|  |  |  |  read symbol in line 11: REAL (3.1415926536)
|  |  |  |  entering expression
|  |  |  |  |  entering constant_literal
|  |  |  |  |  |  read symbol in line 11: SEMICOLON
|  |  |  |  |  leaving constant_literal
|  |  |  |  |  entering expression_prime
|  |  |  |  |  leaving expression_prime
|  |  |  |  leaving expression
|  |  |  leaving constant_declaration
|  |  |  read symbol in line 13: KEY_VAR
|  |  leaving declaration
|  |  entering declaration
|  |  |  read symbol in line 14: IDENT (x)
|  |  |  entering variable_declaration
|  |  |  |  read symbol in line 14: COLON
|  |  |  |  read symbol in line 14: KEY_REAL
|  |  |  |  entering type_denoter
|  |  |  |  |  read symbol in line 14: SEMICOLON
|  |  |  |  leaving type_denoter
|  |  |  leaving variable_declaration
|  |  |  read symbol in line 16: KEY_BEGIN
|  |  leaving declaration
|  |  read symbol in line 17: IDENT (x)
|  |  entering statement_sequence
|  |  |  entering statement
|  |  |  |  entering assignment_statement
|  |  |  |  |  entering variable_designator
```

```
|  |  |  |  |  |  read symbol in line 17: ASSIGN
|  |  |  |  |  |  entering variable_designator_prime
|  |  |  |  |  |  leaving variable_designator_prime
|  |  |  |  |  leaving variable_designator
|  |  |  |  |  read symbol in line 17: INTEGER (2)
|  |  |  |  |  entering expression
|  |  |  |  |  |  entering constant_literal
|  |  |  |  |  |  |  read symbol in line 17: PLUS
|  |  |  |  |  |  leaving constant_literal
|  |  |  |  |  |  entering expression_prime
|  |  |  |  |  |  |  read symbol in line 17: IDENT (pi)
|  |  |  |  |  |  |  entering expression
|  |  |  |  |  |  |  |  entering variable_designator
|  |  |  |  |  |  |  |  |  read symbol in line 17: SEMICOLON
|  |  |  |  |  |  |  |  |  entering variable_designator_prime
|  |  |  |  |  |  |  |  |  leaving variable_designator_prime
|  |  |  |  |  |  |  |  leaving variable_designator
|  |  |  |  |  |  |  |  entering expression_prime
|  |  |  |  |  |  |  |  leaving expression_prime
|  |  |  |  |  |  |  leaving expression
|  |  |  |  |  |  |  entering expression_prime
|  |  |  |  |  |  |  leaving expression_prime
|  |  |  |  |  |  leaving expression_prime
|  |  |  |  |  leaving expression
|  |  |  |  leaving assignment_statement
|  |  |  leaving statement
|  |  |  read symbol in line 18: KEY_END
|  |  |  entering statement
|  |  |  |  entering empty_statement
|  |  |  |  leaving empty_statement
|  |  |  leaving statement
|  |  leaving statement_sequence
|  |  read symbol in line 18: IDENT (HelloWorld)
|  leaving block
|  read symbol in line 18: PERIOD
|  read symbol in line 19: END_OF_FILE
leaving module
```

Durch die Einrückung wird der implizit aufgebaute Syntaxbaum sichtbar. Durch minimale Modifikation wäre z.B. eine Ausgabe im XML-Format möglich.

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 sollen Fehlermeldungen erzeugt werden, falls die Eingabe kein gültiges Modula-2-Programm ist. In solchen Fällen soll das Programm ebenfalls mit einer Fehlermeldung und dem Exit-Code 1 abbrechen.

Für eine fehlerhafte Eingabe `error.m`

```
1  MODULE HelloWorld;
2
3  END .
```

soll sich `moco` wie folgt verhalten:

```
$ ./moco error.m output.txt
unexpected PERIOD, expected IDENT
error in line 3: syntax error
last token: .
```

Die letzten beiden Zeilen der Fehlerausgabe werden über die vorgegebene generische Funktion `void yyerror(char const*)` realisiert. Genauere Fehlermeldungen sind jeweils vom Kontext des Fehlers abhängig und müssen an den jeweiligen Stellen im Parser realisiert werden. Dies ist auch Teil der Zusatzaufgabe.

Im Fehlerfall müssen Sie sich nicht um das Schließen von Dateien oder das Freigeben von Speicher kümmern.

Vorgaben

- Mittels dem gegebenen `Makefile` muss sich das lauffähige Programm `moco` erstellen lassen.
- Beschreibung der einzelnen vorgegebenen Dateien:
  - `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 nur in der Zusatzaufgabe verändert werden. Beachten Sie, dass Sie sich in einem Fehlerfall nicht um das Schließen von Dateien oder das Freigeben von Speicher kümmern müssen.
  - `scanner.l` – Hier soll der 1. Teil der Aufgabe umgesetzt werden. Diese Datei enthält bereits ein grobes Gerüst eines Flex-Scanners mit ein paar vorgegebenen Parametern, sodass mit `flex scanner.l` die Dateien `scanner.c` und `scanner.h` generiert werden. Im Header `scanner.h` werden die Variablen `yytext`, `yylineno`, `yyin`, `yyout` und die Funktionen `int yylex()` (der Scanner) und `int yylex_destroy()` (eine Funktion zur Freigabe des vom Scanner verwendeten Speichers) zur Verfügung gestellt. In der Datei `scanner.c` werden auch die benötigten Header `parser.h` (für den Typ `yytokentype`) und `diagnosis.h` (für die Funktion `yyerror`) eingebunden.
  - `parser.*` – In diesen Dateien soll der 2. Teil der Aufgabe umgesetzt werden. Es sind bereits mehrere Hilfsfunktionen implementiert, die sich um die Kommunikation mit dem generierten Scanner und die Ausgabe der Statusmeldungen kümmern:
    - `typedef enum yytokentype` – Diese Enumeration enthält die Codes der einzelnen Zeichen wie sie in der Sprachreferenz genutzt werden. Diese Codes müssen im Flex-Scanner als Rückgabe verwendet werden. Gegenüber der Sprachreferenz wurde zusätzlich noch ein weiterer Code, `END_OF_FILE`, hinzugefügt, um das Ende der Eingabe zu kennzeichnen.
    - `const char* tokenNames[]` – Dieses Array enthält die Codes als Zeichenketten und wird in der Ausgabe verwendet. Dabei ist z.B. `tokenNames[KEY_IF]` eben `''KEY_IF''`.
    - `yytokentype currentToken` – Diese Variable enthält das zuletzt gelesene Token und wird von der Funktion `void NEXT()` gesetzt.
    - `unsigned int indent_current` und `const unsigned int indent_step` – Diese beiden Variablen steuern die Einrückung in der Ausgabe. Die Variablen werden von den Funktionen `void entering(char* s)` und `leaving(char* s)` beeinflusst.
    - `void indent()` – Rückt die Ausgabe entsprechend ein.
    - `void entering(char* s)` und `leaving(char* s)` – Diese Funktionen sollen zu Beginn bzw. zum Ende jeder Funktion eines Nichtterminalzeichens mit deren Namen (ohne Klammern oder Typ) gerufen werden.
    - `void NEXT()` – Liest das nächste Token vom Scanner und gibt es außerdem mit Typ und Zeilennummer aus.
    - `int TEST(yytokentype s)` – Testet, ob das aktuelle Token dem übergebenen Token entspricht. Falls dies der Fall ist, wird `1` zurückgeben, ansonsten `0`.

    * `void MATCH(yytokentype s)` – Implementiert die Match-Operation des Top-Down-Parsers. Es wird ein gegebenes Token gegen das aktuelle Token gematcht. Falls dies erfolgreich ist (also die Tokens übereinstimmen), wird direkt das nächste Token angefordert. Falls die Token unterschiedlich sind, liegt ein Fehler vor und es wird eine entsprechende Ausgabe auf `stderr` erzeugt und mit `yyerror()` das Programm abgebrochen.
    * `void yyparse()` – Diese Funktion kapselt den eigentlichen Parser und soll in der `main()`-Funktion gerufen werden.

  All diese Funktionen sollen *nicht* verändert werden. Zu implementieren sind für jedes Nichtterminalzeichen eine entsprechend benannte Funktion und gegebenenfalls weitere Hilfsfunktionen (siehe *Hinweise*). Als Beispiel ist die `void module()`-Funktion vorgegeben, wobei der interne Aufruf von `block()` auskommentiert ist, damit das Programm compiliert. Achten Sie darauf, diese Zeile zu entkommentieren.

- Das eigentliche Programm ist bereits in der Datei `main.c` implementiert, wo zunächst die Kommandozeilenparameter ausgewertet werden, dann die Ein- und Ausgabe initialisiert wird, und dann der Parser mit `yyparse()` gerufen wird. Abschließend wird der allokierte Speicher freigegeben und die Dateien geschlossen. Der Scanner wird mit `yylex_destroy()` freigegeben.

Hinweise

- Sie müssen **nicht** die gesamte Sprachreferenz umsetzen. Speziell soll `function_call` weglassen werden. Dementsprechend:
  1. In Regel `statement` soll die Alternative `function_call` **nicht** berücksichtigt werden.
  2. In Regel `expression` soll die Alternative `function_call` **nicht** berücksichtigt werden.
- Flex kümmert sich dank der Zeile `%option yylineno` bereits automatisch um Zeilennummern und aktualisiert ständig die Variable `int yylineno`. Allerdings ist es dafür nötig, dass der zu matchende Whitespace explizit in einer Regel (also nicht implizit über das Catch-all-Pattern `.`) aufgezählt wird, z.B. via

  ```
  [\n\r\t ]   { /* skip whitespace */ }
  ```

  Denken Sie auch an Whitespace in Kommentaren!
- Das Top-Down-Verfahren ist nicht freiwillig deterministisch, da die Auswahl der Regel nicht immer eindeutig ist. Wir wollen *keinen* Parser mit Backtracking implementieren, sondern den Nichtdeterminismus mit FIRST- und FOLLOW-Mengen auflösen. Das notwendige Vorgehen ist in den Vorlesungsfolien beschrieben.
- Ebenfalls problematisch sind Linksrekursionen. Wie diese aufgelöst werden können, ist ebenfalls in den Vorlesungsfolien beschrieben. Im oberen Beispiel wird bspw. die Funktion `expression_prime()` oder `variable_designator_prime` gerufen, die durch eine notwendige Umformung der Grammatik entstanden ist.
- Bei `block` muss die FIRST(`declaration`) beachtet werden. Bei `variable_designator` muss die Linksrekursion beachtet werden und weg-transformiert werden.