Projektdokumentation

22. Januar 2015

Inhaltsverzeichnis

1 Architektur

Der Compiler besteht aus mehreren Komponenten:

- *Buffer* - Zuständig für das Lesen aus einer Datei. Der Buffer implementiert, wie der Name sagt, einen eigenen Zeichenpuffer, d.h. er greift nicht auf dem vom Betriebssystem bereitgestellten Puffer zurück. Deshalb kann der Buffer nur eine begrenzte Anzahl von Zeichen rückwärts lesen - nämlich geneu die, die noch im Puffer sind. Man sollte deshalb möglichst nur neue Zeichen lesen und nicht in der Datei zurückspringen.
- *Token* - Die Token des Compilers.
- *Automat* - Produziert Token aus einem Zeichenstrom. Der Automat besitzt einen internen Zustand, der je nach eingelesenem Zeichen wechseln kann. Jedes an den Automaten übergebene Zeichen kann deshalb einen Seiteneffekt haben - man muss die Ausgabe des Automaten also zwischenspeichern wenn man sie öfter benötigt.
- *Symboltable* - Speichert gefundene Identifier. Um erkennen zu können ob ein Identifier schon ein mal gefunden wurde und ob es sich bei dem Identifier um ein Schlüsselwort handelt werden diese in der Symboltable gespeichert. Sie ermöglicht ein schnelles abspeichern und wiederfinden der Identifier. Außerdem gibt sie ein sogenanntes Symbol zurück, das als Referenz auf einen Identifier dient und zusätzliche nützliche Informationen über den Identifier speichert.
- *ErrorHandler* - Speichert die Position und eine Fehlermeldung zu einem gefundenen Fehler. Der ErrorHandler kann gefundene Fehler zusammen mit ihrer Position, der Fehlermeldung und der Zeile, in der der Fehler vorkam, auf der Konsole ausgeben.
- *Scanner* - Kommuniziert mit Buffer, Automat, Symboltable und ErrorHandler um einem Tokenstrom zu produzieren. Dabei nimmt der Scanner so lange Zeichen aus dem Buffer entgegen und gibt diese an den Automat weiter bis dieser einen Token produzieren kann. Der produzierte Token wird daraufhin an die Komponente, die den Scanner aufruft, weitergegeben. Tritt ein Fehler beim produzieren eines Tokens auf, z.B. weil ein ungültiges Zeichen gefunden wurde, dann wird dieser Fehler im ErrorHandler gespeichert und der Scanner gibt ein Error Token aus. Wird ein Identifier Token erkannt prüft der Scanner mit Hilfe der Symboltable ob es sich um ein Schlüsselwort handelt und gibt dementsprechend ein anderes Token zurück. Sobald der Buffer keine weiteren Zeichen mehr zur Verfügung stellt, gibt der Scanner ein EOF Token aus.

- *Parser* - siehe Dokumentation im Kapitel Parsing
- *Typer* - siehe Dokumentation im Kapitel Typechecking
- *CodeGen* - siehe Dokumentation im Kapitel Codegenerierung

2 Wichtige Verzeichnisse und Dateien

```
|-- Automat
|   |-- makefile
|   |-- src
|-- Buffer
|   |-- makefile
|   |-- src
|-- Scanner
|   |-- makefile
|   |-- src
|-- Symboltable
|   |-- makefile
|   |-- src
|-- sharedlib
|-- doc
|-- makefile
|-- run_tests.sh
|-- tests
```

- Die Verzeichnisse *Automat*, *Buffer*, *Scanner*, *Symboltable* sind auf die Verzeichnisse *debug*, *lib* und *src* aufgeteilt, wobei die ersten beiden automatisch vom *makefile* erzeugt werden wenn sie nicht vorhanden sind.
- *sharedlib* beinhaltet alle gelinkten library Dateien, die zur späteren Ausführung des Compilers benötigt werden.
- *doc* beinhaltet die Projektdokumentation.
- *run_tests.sh* ist die Testsuite. Sie durchsucht das Verzeichnis *tests* nach vorhandenen Tests und führt diese aus.
- *tests* beinhaltet alle Tests.

Um das Projekt zu bauen genügt es das oberste *makefile* auszuführen. Um zu überprüfen ob alles funktioniert sollte nach dem Bauen des Projekts zuerst die Testsuite ausgeführt werden.

Möchte man den Compiler manuell ausführen muss man zuallererst die library Dateien exportieren:

```
$ export LD_LIBRARY_PATH=sharedlib:$LD_LIBRARY_PATH
```

2.1 Testsuite

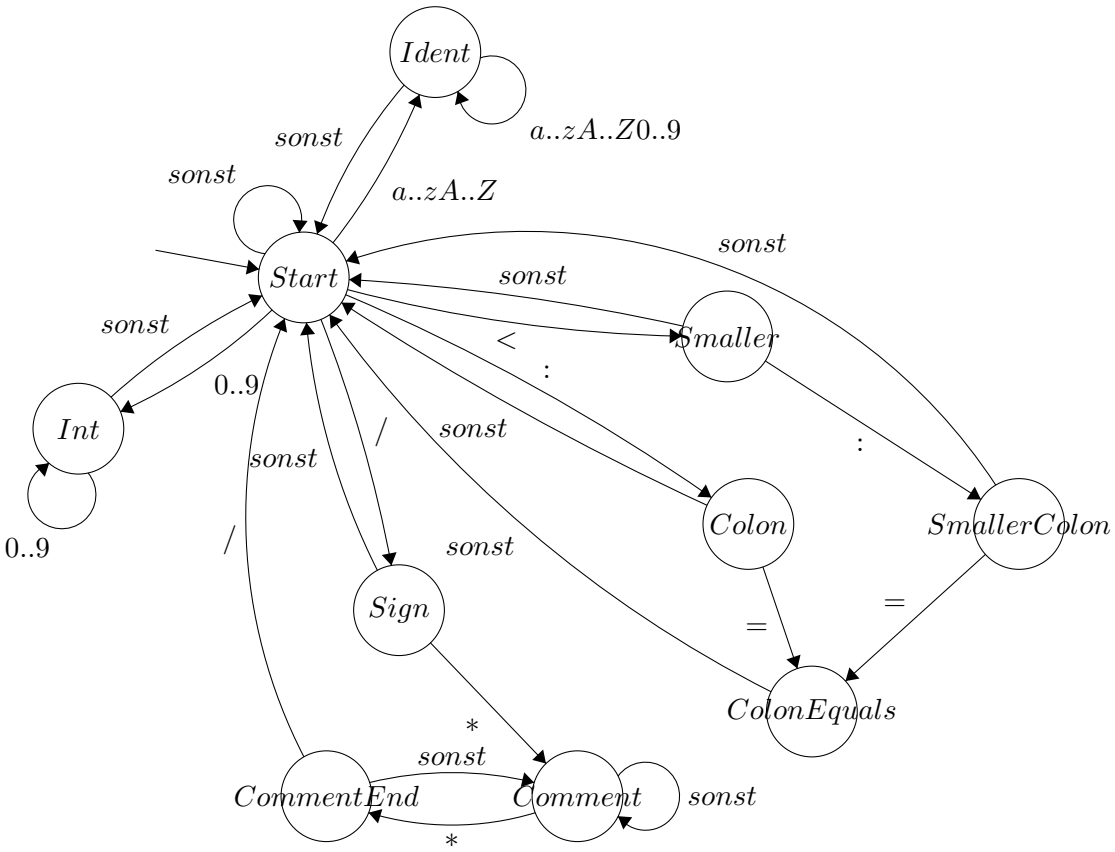
Die Datei *run_tests.sh* verfügt über folgende Funktionalität:

- Alle Dateien in *tests*, die die Endung *.test* besitzen werden als Eingabe für den Compiler verwendet.
- Zu jeder *.test* Datei muss es eine *.check* Datei geben. Diese beinhaltet die Ausgabe des Compilers. Die Testsuite prüft ob der Inhalt dieser Datei mit der tatsächlich produzierten Ausgabe des Compilers übereinstimmt.
- Optional kann noch eine *.flags* Datei angegeben werden. Diese beinhaltet Argumente für die Kommandozeile, die an den Compiler übergeben werden.
- Schlägt ein Test fehl wird eine *.run* Datei angelegt, die die tatsächliche Ausgabe der Compilers beinhaltet.

3 Codedokumentation

3.1 Automat

Die Funktionsweise des Automaten wird durch das folgende Diagramm verdeutlicht:



4 Parsing

Der Parser ist dafür zuständig die Token vom Scanner entgegen zu nehmen um dann daraus einen Syntaxbaum generieren zu können. Dabei überprüft der Parser syntaktische Korrektheit des Programms und gibt Fehlermeldungen aus falls er einen Syntaxfehler findet. Der Parser kann keine semantischen Fehler im Programm finden (z.B. die Zuweisung eines Integers an eine Variable, die ein Array referenziert), dies übernimmt der Typechecker, der aufgerufen wird sobald die Parsephase erfolgreich abgeschlossen wurde.

4.1 Funktionsweise

Der Parser ist ein recursive-descent-parser und parst folgende Grammatik:

```
PROG       ::= DECLS STATEMENTS
DECLS      ::= DECL ; DECLS | eps
DECL       ::= int ARRAY identifier
ARRAY      ::= [ integer ] | eps
STATEMENTS ::= STATEMENT ; STATEMENTS | eps
STATEMENT  ::= identifier INDEX := EXP
             | write ( EXP )
             | read ( identifier INDEX )
             | {STATEMENTS}
             | if ( EXP ) STATEMENT else STATEMENT
             | while ( EXP ) STATEMENT
EXP        ::= EXP2 OP_EXP
EXP2       ::= ( EXP ) | identifier INDEX | integer | - EXP2 | ! EXP2
INDEX      ::= [ EXP ] | eps
OP_EXP     ::= OP EXP | eps
OP         ::= + | - | * | / | < | > | = | <:> | &

eps = epsilon symbol
```

Die obige Grammatik ist bereits eine parsing-expression-grammar (PEG), d.h. Linksrekursion und Mehrdeutigkeiten wurden bereits aufgelöst. Der Parser parst also exakt diese Grammatik und wandelt sie in einen Syntaxbaum um, der exakt der PEG entspricht. Der Parser macht einen Lookup auf das nächste Token bevor er eine Rekursionsstufe

tiefer geht, was allerdings nur passiert wenn er ein validen Token findet. Im Falle eines invaliden Tokens wird eine Fehlermeldung ausgegeben, die anzeigt welche Token erwartet und welches gefunden wurde.

Da der Parser die PEG exakt parst, wächst der Stack sehr schnell - bei auftretenden Überläufen im Parser muss der Stack also erhöht werden. Es ist möglich, die Grammatik so zu transformieren, dass der Stack nicht so stark wächst, dies wurde allerdings nicht umgesetzt. Der commit `050b035c5f717f460ae840574dbe0e8169cba916` enthält ein Beispiel, das zeigt, wie man den Stack in C++ erhöht.

4.2 Parsebaum

Da der Parsebaum die PEG exakt repräsentiert, besitzt er genau so viele unterschiedliche Knotentypen wie es Nichtterminale in der PEG gibt. Die Implementierung stellt aber nur einen Typ names `Node` bereit, der eine Methode namens `tpe` bereitstellt, über die der repräsentierte Typ herausgefunden werden kann. Der `Node` Typ besitzt weitere Methoden, über die man an dessen Inhalt herankommt. All diese Methoden sind abhängig vom repräsentierten Typ, wird versucht auf einen Inhalt zuzugreifen, der nicht repräsentiert wird so wird ein Laufzeitfehler ausgelöst. Ein Beispiel:

```
switch (node->tpe()) {
  case Nodes::Prog:
    node->decls(); // ok
    node->stmts(); // ok
    node->exp(); // Laufzeitfehler
    break;
  case ... // restliche Typen
}
```

Die Namen der Methoden entsprechen in den meisten Fällen den Namen der Nichtterminale in der PEG, welcher Typ welche Operationen unterstützt steht in den Dokumentationskommentaren von `Node`.

Der Baum kann sehr einfach traversiert werden, die Vorgehensweise im Allgemeinen:

```
void traverse(Nodes::Node *node) {
  switch (node->tpe()) {
    case Nodes::Prog:
      traverse(node->decls());
      traverse(node->stmts());
      // behandle decls() und stmts() falls notwendig
      break;
    case ... // restliche Typen
  }
}
```

Die Tests zu der Parserphase lassen sich im Verzeichnis `tests/parser` finden.

5 Typechecking

Beim Typechecking wird überprüft, ob die drei existierenden Typen `Int`, `IntArray` und `NoType` korrekt benutzt wurden und ob referenzierte Variablenbezeichner existieren und ob Operationen auf einem Typ definiert sind. Eine Fehlermeldung wird ausgegeben wenn ein Fehler gefunden wurde. Die Typüberprüfung wurde durch einen einfachen Traverser realisiert, der Baum muss nicht mehrere Male durchlaufen werden, da der erste Durchlauf alle möglichen Fehlermöglichkeiten bereits vollständig auflösen kann. Die gefundenen Typinformationen werden in dem, zum Variablenbezeichner zugehörigen `Symbol` abgelegt, das über den Baum referenziert werden kann.

Die Tests zu der Typisierungsphase lassen sich im Verzeichnis `tests/typer` finden.

6 Codegenerierung

Bei der Codegenerierung müssen keine Fehler im Syntaxbaum gesucht werden, alle möglichen Fehlerquellen wurden bereits aufgelöst. Die Codegenerierung funktioniert sehr einfach. Es ist wieder ein Traverser, der für jeden Knotentyp den zugehören Bytecode für die virtuelle Maschine ausspuckt. Die einzige Schwierigkeit liegt in der Generierung des Codes für die Operationen `> (größer)` und `<:> (ungleich)`, da diese von der virtuellen Maschine nicht unterstützt werden. Sie müssen durch Äquivalenzumformungen mit Hilfe von anderen Operationen repräsentiert werden.

Die Tests zu der Codegenerierungsphase lassen sich im Verzeichnis `tests/codegen` finden.