Phoenix

Konstruktion eines Compilers

Alexander Miller, Daniel Brand

VP Grundlagen Compilerbau
Fachbereich für Computerwissenschaften
Universität Salzburg

11. Juli 2013

Inhaltsverzeichnis

1 Einleitung

Diese Dokumentation beschreibt die Features unseres Compilers Phoenix. Wir entschieden uns für C als Sprache, da die geforderten sprachlichen Features damit unmittelbar umgesetzt werden können.

1.1 Team

Phoenix wurde geschrieben von:

- Daniel Brand (1023077)
- Alexander Miller (1120667)

2 Features

Die Features werden unterstützt:

- Basic Types (**char** und **int**)
- Arrays
- Records
- Boolean Expressions
- Arithmetic Expressions
- Constant Folding
- Strings
- File I/O
- **while** (auch verschachtelt)
- **if**/**else** (auch verschachtelt)
- Lazy Evaluation
- Prozeduren
- Call-by-value (bei Basic Types)
- Call-by-reference (bei Arrays und Records)
- Scoping von Variablen (Global und lokal)
- Type-checking
- Self-Scanning

Nicht implementiert:

- Self-Compilation
- Seperate Compilation

3 Scanner

Der Scanner liest ein Textfile in ASCII-Kodierung ein und fügt Zeichen zu syntaktischen Einheiten (Tokens) zusammen. Diese Tokens sind in ihrer Bedeutung eindeutig definiert (Anhang 1).

Der Scanner in Phoenix liest solange Zeichen von der Eingabedatei ein, bis er eine Wortgrenze erkennt. Danach versucht er, das bis zu diesem Zeitpunkt eingelesene Wort zu kategorisieren und dies als Token an den Parser weiter zu geben.

Abweichend von C haben wir das **static** Keyword benutzt, um frühzeitig die Parsingentscheidung zu treffen, ob eine Variable deklariert wird, oder eine Funktion. Wenn alle Variablen und Funktionen im selben File sind (bzw. die include-Anweisungen direkt auf die .c-Daten referenzieren), kann der Quellcode ohne Modifikationen auch vom GCC übersetzt werden.

4 Parser

Der Parser enthält mit Parsing und Code Generation den größten Teil der Logik. Der Aufbau des Parser orientiert sich an der in der Vorlesung vorgestellten Implementierung. Es ist ein LL(1) recursive descent Parser. Im Anhang 1 ist die verwendete Extended Backus-Naur-Form zu sehen.

4.1 Typen

Phoenix unterstützt die Deklaration von Variablen vom Typ Integer und Character. Eine Zuweisung einer Variable Character muss ebenfalls mit dem ASCII-Wert des Buchstabens geschehen. Eine Zuweisung der Form ch='a' wird nicht unterstützt. Implizit können auch Booleans deklariert werden. Da aber der boolesche Datentyp in C nicht explizit vorhanden ist, kann keine Variable vom Typ bool angelegt werden. Eine Zuweisung eines booleschen Ausdrucks zu einer Integervariable ist jedoch möglich (wird aber vom Parser als Typkonflikt erkannt).

4.1.1 Strings

Strings können nicht nur zur Laufzeit als **char**-Arrays erstellt werden, sondern auch im Quellcode als Konstanten verwendet werden. Das ermöglicht Ausgaben mit printf und die im Scanner häufig benutzten stringCompares mit den definierten Keywords.

string.c

```
1  void printMe(char * string)
2  {
3          printf(string);
4  }
5
6  void main()
7  {
8          printf("Hello");
9          printMe("World");
10 }
```

Ausgabe der Target Machine

```
Phoenix: Margit
================
Loaded 160 bytes

>'Hello'

>'World'

Execution stopped.
```

4.1.2 Arrays

Arrays werden mittels malloc zur Laufzeit am Heap erstellt. Sie können aus den oben genannten Basic Types bestehen.

4.1.3 Records

Wie Arrays werden auch Records (**struct** in C) zur Laufzeit mittels malloc am Heap erzeugt.

Beispiel eines Structs das behandelt werden kann

```
struct type_t;
struct object_t{
        char  *name;
        int class;
        int offset;
        struct type_t *type;
        struct object_t *next;
        struct object_t *previous;
        struct object_t * params;
        int value;
        int reg;
};
```

4.2 Schleifen

Phoenix-C erlaubt nur while-Schleifen. for-Schleifen können aber damit ebenfalls ausgedrückt werden.

4.3 Type Checking

In diesem Beispiel sieht man die schon beschriebene Verwendung von booleschen Ausdrücken und Characters.

types.c

```
1 void main()
2 {
3         int i;
4         char ch;
5
6         i = (1<2); // Boolean expression
7         ch = 65;
8 }
```

Ausgabe des Compilers

```
Phoenix: Parser
===============
2 main

Warning Near Line 6: type mismatch in assignment

Warning Near Line 7: type mismatch in assignment

Parsed with 0 errors, 2 warnings
```

4.4 Boolean Expressions

Boolean Expressions werden üblicherweise als Conditionals in **if**- oder **while**-Konstrukten verwendet. Wie in C haben wir keine Keywords für die konstanten true und false vorgesehen.

4.4.1 Lazy Evaluation

Erlaubt das Evaluieren eines booleschen Ausdrucks solange bis das Ergebnis eindeutig bekannt ist. Eine Konjunktion wird dadurch vorzeitig verlassen, sobald ein Term false ist.

4.5 Arithmetic Expressions

Bei den unterstützen arithmetischen Operationen haben wir uns auf die unbedingt benötigten Funktionen beschränkt.

In arithmetischen Ausdrücken werden konstante Werte zusammengefasst (Constant folding). Aus $x + 3 + 5$ wird so zuerst $x + 8$ bevor Code generiert wird.

4.6 Code Generation

Die Code Generation geschieht während dem Parsing. Die Generierung von Code wird so lange wie möglich verzögert, um Optimierungen wie Constant Folding zu erlauben.

5 Target Machine

Die Target Machine ist eine DLX-Maschine. Somit besitzt sie 32 Register mit jeweils 32bit. Zusätzlich zu den DLX-Befehlen haben wir neue Instruktionen für Input/Output eingeführt. Zusätzlich haben wir das Sichern und

Wiederherstellen von Registern bei Prozeduraufrufen als je eine Instruktion ausgedrückt, was zu einer Reduktion der Größe der Binärdatei des Parser von bis zu 50% führte.

5.1 Binärformat

Das erwartete Binärformat ist in folgende Segmente aufgeteilt:

- 1. Instruktion: TRAP
- 2. Instruktion: Jump to main
- Code
- Strings
- Globale Variablen

Zur Laufzeit werden von der Target Machine Heap und Stack zur Verfügung gestellt.

5.2 Ausführung

Die Target Machine lädt das Binärfile in den virtuellen RAM. Der GP wird an die erste freie Stelle nach dem gelesen File gesetzt. Der PC wird auf 1 gesetzt. Die Semantik der Befehle ist ansonsten weitgehend analog zu denen in der Vorlesung.

5.3 System IO

Wir haben neue Instruktionen für Input und Output eingeführt. Diese sind analog zu den benutzten POSIX-Syscalls.

- fopen
- fclose
- fgetc
- fputc
- printf

Die Target Machine verwaltet ein Array an geöffneten Dateien und gibt dem Programm nur den Index zurück. Damit umgeht man die Probleme von Zuweisungen von 64bit Filepointern zu 32bit Registern.

6 Conclusio

Die abschließenden Tests haben ergeben, dass der Compiler self-scanning ist (siehe final-Milestone). Der Scanner, in der Target Machine ausgeführt, erkennt sowohl die Tokens im scanner sowie im Parser. Der ausgeführte Parser jedoch hat einen Fehler beim Erkennen der Tokens. Höchstwahrscheinlich liegt es an der Erstellung von Strings oder an der Speicherung von konstanten Strings, welche im Scanner benötigt werden.

Seperate Compilation haben wir nicht implementiert, stattdessen wurden die aufgeteilten Files konkateniert.

7 Anhang 1: EBNF

```
start = {include_def} {top_declaration}.

include_def = "#include" (string_literal ||
("<" {identifier || "."} ">") ).

top_declaration = type_declaration ";" ||
variable_declaration ";" || function_declaration.

type_declaration = struct_declaration ||
typedef_declaration.

variable_declaration = ["static"] type
["*"] identifier.

function_declaration = type identifier
formalParameters (";" || "{"
[variableDeclarationSequence] {instruction} "}").

struct_declaration = "struct" identifier
"{" {variable_declaration ";"} "}".

type = "int" || "char" || "void" || identifier ||
("struct" identifier).

identifier = letter {letter || digit}.

formalParameters = "(" formalParameter { ","
formalParameter } ")".

formalParameter = type identifier.

variableDeclarationSequence =
{ variable_declaration ";" }.

instruction = if_else || fclose_func ";" ||
while_loop ||  return_statement ";" ||
printf_func ";" || fputc_func ";" || identifier
( actualParameters || "=" expression) ";".
```

```
if_else = "if" "(" expression ")" "{"
{ instruction } "}" [ "else" "{"
{ instruction } "}" ].

fclose_func = "fclose" "(" expression ")".

while_loop = "while" "(" expression ")" "{"
{ instruction } ")".

return_statement = "return" [expression].

printf_func = "printf" "(" expression ")".

fputc_func = "fputc" "(" expression ","
expression ")".

actualParameters = "(" [expression
 {"," expression } ] ")".

expression = simple_expression
[ ("==" || "<=" || "<" || "!=" || ">" || ">=")
expression ].

simple_expression = ("-" simple_expression) ||
term [ ("+" || "-" || "||") term].

term = factor [ ("*" || "/" || "&") factor].

factor = ("!" factor) || "(" expression ")" ||
integer || string_literal || identifier ||
sizeof_func || malloc_func || fopen_func ||
fgetc_func || fputc_func.

sizeof_func = "sizeof" "(" type ")".

malloc_func = "malloc" "(" expression ")".

fopen_func = "fopen" "(" expression
["," expression] ")".

fgetc_func = "fgetc" "(" expression ")".
```

```
fputc_func = "fputc" "(" expression ","
expression ")".

string_literal = '"' string '"'.

string = letter { letter || digit || "_" }.

integer = digit { digit }.

letter = A–Za–z.

digit = 0–9.
```

8 Anhang 2: Unterstütze Instruktionen

```
NOP     // Initialwert im Memory
// F1 (1-23)
ADDI    // Addition mit einem konstanten Wert
SUBI    // Subtraktion
MULI    // Multiplikation
DIVI    // Division
MODI    // Modulo
CMPI    // Vergleich mit einem konstanten Wert
LW      // Laden in ein Register
SW      // Speichern in Speicherstelle
POP     // Wert vom Stack holen
PSH     // Wert auf dem Stack speichern
BEQ     // Branch wenn Register gleich 0
BGE     // Branch wenn groesser oder gleich
BGT     // Branch wenn groesser
BLE     // Branch wenn kleiner oder gleich
BLT     // Branch wenn kleiner
BNE     // Branch wenn ungleich
BR      // Branch
BSR     // Branch in Subroutine
MALLOC  // Allokieren am Heap
RET     // Zurück zur aufrufenden Prozedur
FOPEN   // File oeffnen
FGETC   // Character lesen
FPUTC   // Character schreiben
// F2 (24-43)
SUB     // Subtraktion von Registern
MUL     // Multiplikation
DIV     // Division
MOD     // Modulo
CMP     // Compare
AND     // Boolesche Konjunktion
OR      // Boolesche Disjunktion
PRINTF  // Ausgabe eines Strings
PRINTFI // Ausgabe einer Zahl
ADD     // Addition von Registern
// F3 (43-63)
JSR     // Jump in Subroutine
J       // Jump
TRAP    // Erfolg
FCLOSE  // File schliessen
```