technische universität
dortmund

Virtualisierung und Compilation

Übung 3
(ANTLR-Projekt)

Oliver Rüthing

Lehrstuhl für Programmiersysteme

1. Beschreibung

In dem Projekt soll ANTLR für die Syntaxanalyse einer kleinen domänenspezifischen Programmier-sprache eingesetzt werden. Die dabei verwendete Programmiersprache MGPL (Mini Game Programming Language) erlaubt es einfache Arcade-Spiele wie Pong oder Space Invaders zu spezifizieren. Bei MGPL handelt sich um eine stark abgespeckte Variante der Sprache GPL (Game Programming Language), die in einem größer angelegten Compiler-Projekt an der California State University, Chico, verwendet wurde.

Die Startregel von MGPL-Programmen (in EBNF-Notation) macht die generelle Struktur der spezifizierten Spiele ersichtlich:

Prog ::= game Idf (AttAssList ?) Decl* StmtBlock ? Block*

Dem Schlüsselwort game folgt der Name des Spiels, der in der Titelzeile des Spielefensters angezeigt wird. Optional können Attribute des Spieles wie window_width gesetzt werden. Es folgt ein Deklarationsteil, in dem sowohl Variablen als auch graphische Objekte des Spieles definiert werden können. Wir beschränken uns hier auf Integer-Variablen, Kreise, Dreiecke und Rechtecke. Auch Arrays über Variablen und Objekten können gebildet werden, hier allerdings nur mit konstant vorgegebener Größe. Auf den Deklarationsteil folgt ein optionaler Initialisierungsteil, in dem beliebige Anweisungen ausgeführt werden können. Dieser Teil wird einmalig vor Beginn des eigentlichen Spielablaufs ausgeführt.

Das Kernstück des Spiels wird durch eine Sequenz von Animations- und Eventblöcken festgelegt. Animationsblöcke beschreiben, wie graphische Objekte in regelmäßigen Abständen verändert werden sollen. Durch Angabe eines `animation_block`-Attributs wird für ein graphisches Objekt ein entsprechender Animationsblock festgelegt. Beispielsweise legt die Objektdeklaration

```
circle ball(x = 100, y = 200, radius = 5, animation_block = move_ball);
```

ein graphisches Kreisobjekt mit Namen `ball` an, dessen Animation vom Animationsblock `move_ball` übernommen wird. Ein solcher Animationsblock muss dann natürlich vorhanden sein. Findet sich z.B. eine Deklaration wie

```
animation move_ball (circle cur_ball){
          ....     // Statements
    }
```

so werden die Anweisungen in regelmäßigen Abständen auf dem Objekt `ball` ausgeführt. Der Parameter `cur_ball` ist dabei Referenzparameter für das aufrufende Objekt `ball`. Eventblöcke spezifizieren, wie das Spiel auf Tastatureingaben reagiert. Beispielsweise wird durch

```
on leftarrow{
          ....       //  Statements
}
```

festgelegt, welche Anweisungen nach Drücken der linken Pfeiltaste auszuführen sind.

2. Grammatik

Eine genauere Beschreibung der MGPL-Semantik wird zunächst noch zurückgestellt, da es zunächst vornehmlich um Syntaxanalyse geht. Die maßgebliche EBNF von MGPL ist wie folgt:

```
Prog        ::=  game Idf ( AttrAssList ? ) Decl* StmtBlock Block*
Decl        ::=  VarDecl ; | ObjDecl ;
VarDecl     ::=  int Idf Init ? | int Idf [ Number ]
Init        ::=  = Expr
ObjDecl     ::=  ObjType Idf ( AttrAssList ? ) | ObjType Idf [ Number ]
ObjType     ::=  rectangle | triangle | circle
AttrAssList ::=  AttrAss , AttrAssList | AttrAss
AttrAss     ::=  Idf = Expr
Block       ::=  AnimBlock | EventBlock
AnimBlock   ::=  animation Idf ( ObjType Idf ) StmtBlock
EventBlock  ::=  on KeyStroke StmtBlock
KeyStroke   ::=  space | leftarrow | rightarrow | uparrow | downarrow
StmtBlock   ::=  { Stmt* }
Stmt        ::=  IfStmt | ForStmt | AssStmt ;
IfStmt      ::=  if ( Expr ) StmtBlock ( else StmtBlock )?
ForStmt     ::=  for ( AssStmt ; Expr ; AssStmt ) StmtBlock
AssStmt     ::=  Var = Expr
Var         ::=  Idf | Idf [ Expr ] | Idf . Idf | Idf [ Expr ] . Idf
Expr        ::=  Number | Var | Var touches Var  | - Expr | ! Expr | ( Expr ) | Expr Op Expr |
Op          ::=  || | && | == | < | <= | + | - | * | /
```

In der MGPL-Grammatik sind Terminalzeichen grün hervorgehoben. Für die Nichtterminale *Number* und Idf , die blau gekennzeichnet sind, wird auf die Angabe der Produktionen verzichtet. Die Erkennung dieser Symbole soll durch die Angabe entsprechender Scanner-Regeln erfolgen. Dabei gilt:

- Identifikatoren in Idf sind Folgen aus Buchstaben, Ziffern und dem Sonderzeichen '_ ', die mit einem Buchstaben beginnen.
- Zahlen in Number sind beliebige Ziffernfolgen. Negative Zahlen sind hier nicht vorgesehen, können aber durch Verwendung des unären Negationsoperators leicht erzeugt werden.

2. Scanner und Parser

Schreiben Sie eine ANTL-Grammatikbeschreibung MGPL.g.

Beachten Sie, dass es sich bei der MGLP-Grammatik nicht um eine eindeutige Grammatik handelt. So sind die Prioritäten der Operatoren durch die Produktionen nicht festgelegt. Hier gilt es durch Einführen weiterer Nichtterminale wie *BoolExpr, RelExpr,…* Prioritäten zu berücksichtigen. Dabei sind die Prioritätsstufen in absteigender Reihenfolge:

- Unäre Operatoren: - und !
- Multiplikative Operatoren: * und /
- Additive Operatoren: + und –
- Relationale Operatoren: ==, <= und <
- Konjunktion: &&
- Disjunktion: ||

In MGPL-Programmen sind Zeilenkommentare der Form

```
//  Kommentar
```

erlaubt. Beim Auftreten von `//` soll der Rest der Zeile vom Scanner ignoriert werden.

Ihre ANTLR-Grammatik soll eine LL(*)-Grammatik sein, die ohne Backtracking arbeitet (ANTLR Option `backtrack=false;`). Besonders schön wäre eine Grammatik, die mit festem kleinen lookahead (ANTLR-Option `k= ..;`) parsebar ist. Beachten Sie, dass ggf. Produktionen (wie z.B. für *Var*) anzupassen sind wenn die LL(k)-Eigenschaft verletzt ist.

Testen Sie den lauffähigen Parser mithilfe der MGPL-Dateien Pong.mgpl und Invaders.mgpl.

Beachten Sie: Parser-Produktionen in ANTLR verwenden Kleinbuchstaben für Nichtterminale, Scanner-Produktionen verwenden Großbuchstaben. Hilfsproduktionen des Scanners (nicht Token-erzeugend) müssen mit `fragment` gekennzeichnet werden.

3. Abstrakter Syntaxbaum

Erweitern Sie eine ANTL-Grammatikbeschreibung MGPL.g zu einer Beschreibung MGPL_AST.g, in der zusätzlich ein abstrakter Sytaxbaum konstruiert wird (ANTLR Option `output=AST;)`. Der AST soll dabei insbesondere

- unnötige Terminalsymbole (wie z.B. Klammern) ausblenden
- Kettenproduktionen (wie z.B. in arithmetischen Ausdrücken) kompaktifizieren
- Die Linksassoziativität von Operatoren wie Subtraktion und Division widergeben
- sinnvolle Knotennamen einführen (z.B. durch Verwendung imaginärer Token).

Abgabe

Bitte schicken Sie das Projekt (inklusive der ANTLR-Dateien, ggf. Screenshots aus ANTLRWorks) als Archiv verpackt bis Mittwoch, den 20.12. 2023 an:

oliver.ruething@tu-dortmund.de

Gruppenabgaben (2-3 Studierende) sind möglich und erwünscht.