Projektpraktikum

Eine Umgebung zum Erleichterten Prototyping von Krypto-Verfahren

Alexander O. Ortner
Universität Klagenfurt
Informatik – Systemsicherheit

Ziel des Projektpraktikums

- Entwicklung einer Programmiersprache für endliche algebraische Strukturen. Das Operieren in endlichen Körpern, Polynomringen und Restklassen soll möglichst ohne expliziten Rückgriff auf Bibliotheksfunktionen, ermöglicht werden.
- Ursprünglich: darauf basierenden Compiler der diese Sprache einliest, analysiert und in ausführbaren Java-Code übersetzt.
- Realisiert: darauf basierenden Interpreter der diese Sprache einliest, analysiert und interpretiert.

- Vorteile die sich ergeben sind:
  - Vereinfachtes Handhaben von Algebraischen Strukturen.
  - Kein expliziter Zugriff auf Bibliotheksfunktionen nötig.
  - „Theorienahe“ Programmierung.

Resultate

- Programmiersprache namens FFapl (Finite Field Application Language) .
- Parser und Interpreter der in FFapl geschriebene Programme einliest, analysiert, und interpretiert.
- Eine integrierte Entwicklungsumgebung für FFapl-Entwickler namens Sunset.
- NSIS (Nullsoft Scriptable Install System) Windows Installationsprogram für Sunset.

FFapl – Finite Filed Application Language

- Langzahl- und Modulo-Arithmetik.
- Operieren mit endlichen Algebraischen Strukturen wie Restklassen, Polynomringen und endlichen Körpern.
- Erstellung von Zufallsgeneratoren: Pseudozufallsgeneratoren und Schnittstelle für echte Zufallsgeneratoren.
- Boolsche-Operatoren: Konjunktion, Disjunktion, NOT.
- Vergleichs-Operatoren: ==, <=, >=, !=, <, >
- Kontrollstrukturen: While- und For- Schleifen.
- Bedingte Anweisungen/Verzweigungen: If-Anweisung und Else-Zweig
- Deklaration von Funktionen und Prozeduren $\rightarrow$ Gewährleistung von Modularität und Wiederverwendbarkeit.
- Verwaltung Typengleicher Variablen mittels Array.
- Verbund verschiedener Datentypen mittels Record.
- Globale Konstanten- und Lokale Variablendeklaration.
- Vordefinierte Funktionen und Prozeduren.
- Verwendung von Kommentaren.

Datentypen

- **String** ... Zeichenkette
- **Boolean** ... Boolscher Wert
- **Integer** ... Langzahl
- **Prime** ... Primzahl
- **Polynomial** ... Polynom
- **Z**(p) ... Restklasse modulo p
- **Z**(p)[x] ... Polynomring modulo p
- **GF**(p,ply) ... Galoiskörper mit Charakteristik p und irreduziblen Polynom ply

Spezielle Datentypen - Zufallsgeneratoren

- Werden auf syntaktischer Ebene nicht von anderen Datentypen unterschieden.
- Nur Lesezugriff.
- Bei jedem Lesezugriff wird Zufallszahl erzeugt.
- Arten:
  - **PseudoRandomGenerator**(seed, max)
    Pseudozufallszahlengenerator mit Startwert seed. Liefert Pseudozufallszahlen zwischen 0 und max.
  - **RandomGenerator**(max)
    Liefert Zufallszahlen zwischen 0 und max. Schnittstelle zu echten Zufallszahlengenerator vorbereitet.
  - **RandomGenerator**(min, max)
    Liefert Zufallszahlen zwischen min und max. Schnittstelle zu echten Zufallszahlengenerator vorbereitet.

Konstanten- und Variablendeklaration

- Globale Konstanten:
  - Nur Lesezugriff
  - Beispiele:

```
const p : Prime := 2;
const ply : Polynomial := [1 + x + x^2];
const gf : GF(p, ply) := [1 + x];
```

- Lokale Variablen:
  - Schreib und Lesezugriff
  - Variablen shadowing: lokale Variablen verbergen globale Konstanten.
  - Beispiele:

```
a,b : Z(3)[x]; //Polynomring modulo 3
primG : PseudoRandomGenerator(5, 100);
ply : Polynomial; // verbirgt globale Konstante ply
f : Z(3)[][]; // zwei-dimensionales Array
r : Record a: Integer; b: Polynomial; EndRecord;
u : Z(3)[x];
```


Funktionen und Prozeduren

- Funktionen verfügen über einen Rückgabetyp Prozeduren hingegen nicht.
- Rekursion und Überladen von Funktionen und Prozeduren.
- Unterstützung von Referenzparametern und Wertparametern.
- Beispiele:

```
//Funktion
function func(val : Integer) : Integer{
        ...
        return ... ;
}
//überladene Funktion
function func(val : Polynomial) : Z()[x]{
        ...
        return ... ;
}
//Prozedur
procedure proc(val : Integer){ ... }
```

Parser – Symboltabellen-Visitor – Interpreter-Visitor

Parser:

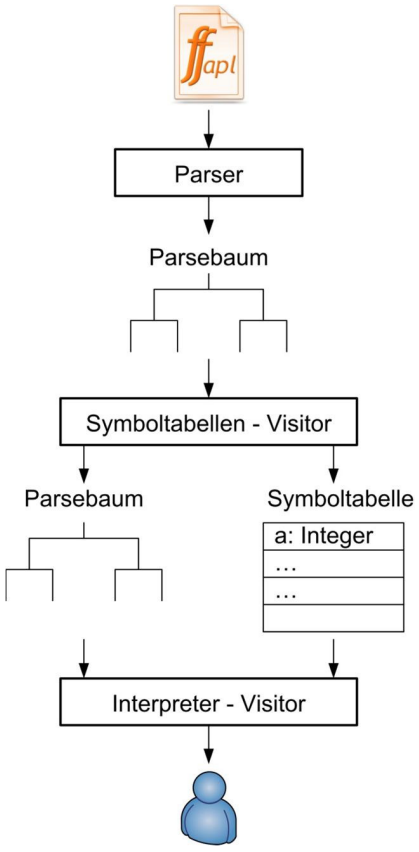
- wurde mit Hilfe des Parsergenerators JavaCC entwickelt.
- Liest FFapl-Datei ein und analysiert die Syntax.
- Erstellt Parsebaum welcher Schnittstelle für Besucher bereitstellt (Visitor-Pattern).

Symboltabellen-Visitor:

- Besucht Parsebaum und erstellt Symboltabelle.
- Führt Typenkontrolle durch (Semantische Analyse).

Interpreter-Visitor:

- Interpretiert Code anhand von Parsebaum und Symboltabelle.
- Führt erweiterte Typenkontrolle durch.



Fehlerausgaben

- Unterstützung von Mehrsprachigkeit (Deutsch und Englisch).
- Einfache Lokalisierung des Fehlers durch Angaben von Zeilen und Spalten.
- Beispiel:

```
program calculate{
        r : Z(6);
        r := 4^-1;
}
```

wirft folgende Fehlermeldung in Deutscher Sprache:

```
FFapl Kompilierung: [calculate] Algebraic Error 106 (Zeile
3, Spalte 15)
Es existiert kein multiplikatives Inverses für 4 in Z(6)
```

wirft folgende Fehlermeldung in Englischer Sprache:

```
FFapl compilation: [calculate] Algebraic Error 106 (line 3,
column 15)
there exists no multiplicative inverse for 4 in Z(6)
```

Fehlerausgauben

- Parser liefert erwartete Syntax.
- Beispiel:

```
program calculate{
        r : Z(3)
}
```

wirft beispielsweise:

```
FFapl Kompilierung: [calculate] ParseException 102 (Zeile 3,
Spalte 1)
"}" erkannt in Zeile 3, Spalte 1. Erwartete jedoch einen
aus:
    "[" ...
    ";" ...
    "[" ...
```


Sunset

- Integrierte Entwicklungsumbegung für FFapl.
- Funktionalitäten umfassen:
  - Datei- bzw. Speichermanagement
  - Undo/Redo - Funktionalitäten
  - Mehrsprachigkeit
  - Syntaxhervorhebung und Fehlerhervorhebung
  - Ausführung von FFapl-Programmen in eigenem Thread
  - Abbruchfunktionalität von laufenden Programmen
  - Individuelle Konsole für jede geöffnete FFapl-Datei
  - Integrierte FFapl-API für Datentypen, vordefinierte Funktionen und Prozeduren, und Beispielimplementierungen
  - Drag- und Drop-Funktionalitäten (öffnen von Dateien, FFapl-API)
  - Shortcut-Key-Funktionalität



Sunset – NSIS – Windows-Installer

- Mehrsprachiger Windows x86 Installer.
- Überprüft installierte Java Version.
- Registriert .ffapl-Dateien im System und verknüpft diese mit Sunset.

Anhang - Syntax

Syntax I

```
letter = 'A' ... 'Z' | 'a' ... 'z' | '_' .
digit = '0' ... '9' .

ident = letter { letter | digit } .
number = digit { digit } .
string = '"' { [^"] } '"'

Random = 'Random' [ '(' Expr [ ':' Expr ] ')' ] .

RelOp = '<=' | '>=' | '>' | '<' .
EqualOp = '==' | '!=' .
AddOp = '+' | '-' .
MulOp = '*' | '/' | 'MOD' .

Literal = 'true' | 'false' | number | Random |
          Polynomial | string .
Selector = '[' Expr ']' [ Selector ] |
           '.' ident [ Selector ] .
```


Syntax II

```
ArrayLen = '#' ident [ Selector ] .

PrimaryExpr = Literal | '(' Expr ')' | ProcFuncCall |
              ident [ Selector ] | ArrayLen |
              '{' ArgumentList '}' .
UnaryExpr = [ AddOp | '!' ] PrimaryExpr .
PowExpr = UnaryExpr { Power UnaryExpr } .
MulExpr = PowExpr { MulOp PowExpr } .
AddExpr = MulExpr { AddOp MulExpr } .
RelExpr = AddExpr [ RelOp AddExpr ] .
EqualExpr = RelExpr [ EqualOp RelExpr ] .
CondAndExpr = EqualExpr { 'AND' EqualExpr } .

CreationExpr = 'new' ArrayType '[' Expr ']'
               { '[' Expr ']' } .

Expr = CondAndExpr { 'OR' CondAndExpr } |
       CreationExpr .
```

Syntax III

```
ArgumentList = Expr { ',' Expr } .
ProcFuncCall = ident '(' [ ArgumentList ] ')' .
Assignment = ident [ Selector ] ':=' Expr .

Condition = '(' Expr ')' .
Block  = '{' { Decl } StatementList '}' .
FuncBlock = '{' { Decl } StatementList ReturnStatement
            '}' .
ElseBlock = IfStatement | Block .
IfStatement = 'if' Condition Block
              [ 'else' ElseBlock ] .
WhileSatement = 'while' Condition Block .
ForStatement = 'for' ident '=' Expr 'to' Expr
               ['step' Expr] Block .
ReturnStatement = 'return' Expr ';' .

StatementList = { Statement ';' | BlockStatement} .
Statement = Assignment | ProcFuncCall .
```


Syntax IV

```
BlockStatement = IfStatement | WhileStatement |
                 ForStatement .

PrimitiveType = 'Integer' | 'Boolean' | 'String'.
ContainerType = Record .
AlgebraicType = 'Prime' | 'Polynomial' .

ComplexAlgebraicType = 'GF' |
                       'Z' '(' ')' [ '[' 'x' ']' ] .
  RandomGeneratorType = 'PsRandomGenerator' |
                      'RandomGenerator' .

ExprRandomGType = PsRandomGenerator | RandomGenerator .
ExprComplexAType = GF |
                   'Z' '(' Expr ')' [ '[' 'x' ']' ] .

ConstType = ExprComplexAType | PrimitiveType |
            AlgebraicType .
```

Syntax V

```
DeclType = ExprRandomGType | ExprComplexAType
           { '[' ']' } | PrimitiveType { '[' ']' } |
           AlgebraicType { '[' ']' } | ContainerType .
ParamType = RandomGType | ComplexAlgebraicType
            { '[' ']' } | PrimitiveType { '[' ']' } |
            AlgebraicType { '[' ']' } .
ArrayType = PrimitiveType | AlgebraicType |
            ComplexAlgebraicType .

Decl = ident { ',' ident } ':' DeclType ';' .
ConstDecl = 'const' ident ':' ConstType ':=' Expr ';' .

IdTerm = 'x' [ '^' PrimaryExpr] .
Term = PowExpr [ IdTerm ] | ['-'] IdTerm .
Polynomial = '[' Term { AddOp  Term } ']' .

GF = 'GF' '(' Expr ',' Expr ')' .
Record = 'Record' { Decl } 'EndRecord' .
```


Syntax VI

```
PsRandomGenerator = 'PseudoRandomGenerator'
                    '(' Expr ',' Expr ')' .
RandomGenerator = 'RandomGenerator'
                  '(' Expr [ ':' Expr ] ')' .

FormalParam = ident { ',' ident } ':' ParamType .
FormalParamList = FormalParam { ';' FormalParam } .

Program = 'program' ident '{' { ConstDecl }
          { Proc | Func } { Decl } StatementList '}' .

Proc = 'procedure' ident '(' [ FormalParamList ] ')'
       Block .
Func = 'function' ident '(' [ FormalParamList ] ')'
       ':' ParamType FuncBlock .
```