PR Übersetzerbau 621.401 - 621.402 SS 2007
Institut für Informationstechnologie M. Taschwer

Implementierung des $YAPL_4$ Compilers

Dokumentversion 1.0

Zeitplan

Thema	Aufgaben	zu bearbeiten bis	Pflichtabgabetermin
Einführung in JavaCC Lexikalische Analyse	1, 2, 3	KW 11	
Ausdrücke in $YAPL_4$ LL(n) Parsing mit JavaCC	4, 5	KW 12	
Syntaktische Analyse	6	KW 13	**30.3. 9 Uhr**
Scoping	7	KW 16	**20.4. 9 Uhr**
Typüberprüfung Typüberprüfung	8 a, b 8 c, d	KW 17 KW 19	 **13.5. 24 Uhr**
Codegenerierung Codegenerierung Codegenerierung Codegenerierung	9 a–c 9 d–f 9 g–i 9 j, k	KW 20 KW 21 KW 23 KW 25	 **24.6. 24 Uhr**

Aufgabenstellung im Überblick

Für die einfache prozedurale Sprache $YAPL_4$ (Yet Another Programming Language, Version 4) ist ein Compiler zu implementieren, der von $YAPL_4$ nach MIPS Assembler-Code übersetzt. $YAPL_4$ stellt ganzzahlige und boolsche Werte sowie Arrays als Datentypen zur Verfügung. Komplexe Anweisungen lassen sich durch Komposition, Verzweigung und while-Schleifen aufbauen. Die Deklaration von Prozeduren ist insofern eingeschränkt, dass keine Forward-Deklarationen und keine geschachtelten Prozeduren erlaubt sind. Die $YAPL_4$ Syntax sowie die vordefinierten Prozeduren, welche von der Laufzeitumgebung zur Verfügung gestellt werden sollen, sind weiter unten angegeben.

Der $YAPL_4$ Compiler ist mit Hilfe von JavaCC (Java Compiler-Compiler) zu entwickeln. Für das Übersetzen und Ausführen des Compilers ist eine Apache Ant Build-Datei bereitzustellen.

Die Aufgaben ab Seite 4 dieser Aufgabenstellung sollen Sie schrittweise zur Entwicklung Ihres $YAPL_4$ Compilers hinführen, wobei der oben angeführte Zeitplan einzuhalten ist. Das Ergebnis (Source Code) von Aufgaben, die mit einem Pflichtabgabetermin versehen sind, müssen dem LV-Leiter per E-Mail fristgerecht übermittelt werden. Pflichtabgaben müssen den Anforderungen ab Seite 3 genügen, damit sie akzeptiert werden können.

Syntax der Sprache $YAPL_4$ in EBNF

```
letter = "A" .. "Z" | "a" .. "z" | "_" .
digit = "0" .. "9" .
otherchar = "+" | "-" | "*" | "/" | "." | "," | ";" | ":" | "!"
          | "?" | "=" | "#" | "%" | "<" | ">" | "$" | "(" | ")"
          | "[" | "]" | "{" | "}" | "\" | "@" | "&" | "^" | "|" .
```

```
ident = letter { letter | digit } .
number = digit { digit } .
string = '"' { " " | letter | digit | otherchar } '"' .

RelOp = "<" .
EqualOp = "==" .
AddOp = "+" | "-" .
MulOp = "*" | "/" .

Literal = "TRUE" | "FALSE" | number .
Selector = "[" Expr "]" [ Selector ] .
PrimaryExpr = Literal | "(" Expr ")" | ProcedureCall
            | ident [Selector] .
UnaryExpr = [AddOp] PrimaryExpr .
MulExpr = UnaryExpr { MulOp UnaryExpr } .
AddExpr = MulExpr { AddOp MulExpr } .
RelExpr = [ "NOT" ] AddExpr [ RelOp AddExpr ] .
EqualExpr = RelExpr [ EqualOp RelExpr ] .
CondAndExpr = EqualExpr { "AND" EqualExpr } .
Expr = CondAndExpr { "OR" CondAndExpr } .

ArgumentList = Expr { "," Expr } .
ProcedureCall = ident "(" [ ArgumentList ] ")" .
Assignment = ident [ Selector ] ":=" Expr .
IfStatement = "IF" Expr "THEN" Statement [ "ELSE" Statement ] .
WhileStatement = "WHILE" Expr "DO" Statement .
ReturnStatement = "RETURN" [ Expr ] .
WriteStatement = "WRITE" string .
Statement = IfStatement | WhileStatement | ReturnStatement
            | WriteStatement | Assignment | ProcedureCall | Block .
StatementList = Statement { ";" Statement } [ ";" ] .
Block = { Decl } "BEGIN" [ StatementList ] "END" .

PrimType = "INTEGER" | "BOOLEAN" .
ArrayDim = "[" ( number | ident ) "]" .
ArrayType = "ARRAY" ArrayDim { ArrayDim } "OF" PrimType .
Type = PrimType | ArrayType .

ConstDecl = ident "=" Literal ";" .
VarDecl = ident { "," ident } ":" Type ";" .
Decl = "CONST" { ConstDecl } | "VAR" { VarDecl } .

FormalParam = ident { "," ident } ":" [ "CONST" ] Type .
FormalParamList = FormalParam { ";" FormalParam } .
Procedure = "PROCEDURE" ident "(" [ FormalParamList ] ")"
            [ ":" PrimType ] Block ident ";" .

ProgramBlock = { Decl | Procedure } "BEGIN" [ StatementList ] "END" .
Program = "PROGRAM" ident ProgramBlock ident "." .
```

Vordefinierte Prozeduren

```
PROCEDURE writeint(i: INTEGER);       /* write i to stdout */
PROCEDURE writebool(b: BOOLEAN);      /* write b ("TRUE" or "FALSE") to stdout */
PROCEDURE writeln();                  /* write a newline character to stdout  */
PROCEDURE readint(): INTEGER;         /* read an integer value from stdin;
                                         characters following the number up
                                         to newline are ignored. */
```

Anforderungen an die Pflichtabgaben

Jede Pflichtabgabe muss den folgenden Anforderungen genügen, damit sie akzeptiert wird:

- Die Abgabe erfolgt in Form einer *einzigen* ZIP-Datei, welche per E-Mail *vor dem Pflichtabgabetermin* an den LV-Leiter gesendet wird.
- Die ZIP-Datei enthält den vollständigen Quellcodebaum der betreffenden Compilerversion, jedoch *ohne* vorkompilierte Java-Klassendateien bzw. Dateien, die von JavaCC erzeugt werden (notwendige Ausnahmen sind erlaubt).
- Das Wurzelverzeichnis des Quellcodebaums enthält eine *build.xml* Datei, welche folgende Kommandozeilenaufrufe unterstützt:

  `ant -Djavacc=<javacc-home>`
  Übersetzt den $YAPL_4$ Compiler unter Verwendung des JavaCC-Installationsverzeichnisses `<javacc-home>`. (Dies ist *nicht* der Pfad zur `javacc` Binärdatei!)

  `ant run -Djavacc=<javacc_home> -Dyapl=<yapl_source> -Doutfile=<asm_output>`
  Führt den $YAPL_4$ Compiler mit der Eingabedatei `<yapl_source>` ($YAPL_4$ Quellcode, absoluter Pfad) aus und speichert den erzeugten Assemblercode in der Datei `<asm_output>` (absoluter Pfad). Das `-Doutfile` Argument muss nur von der Compilerversion *Codegenerierung* unterstützt werden.
- Die `ant` Aufrufe zum Übersetzen und Ausführen des $YAPL_4$ Compilers müssen sämtliche Compilermeldungen (inkl. Fehlermeldungen) auf dem Standardausgabe- bzw. Standardfehlerausgabekanal (`System.out` bzw. `System.err`) ausgeben.
- Der Compiler muss zumindest jene $YAPL_4$ Testdateien, deren Name in `min.yapl4` endet, richtig verarbeiten. $YAPL_4$ Testdateien sind auf der Praktikumshomepage verfügbar.
- Der abgegebene Quellcode muss mit den in Aufgabe 1 angegebenen Versionen von Java, JavaCC und ANT fehlerfrei übersetzt und ausgeführt werden können.

1. Installation der Entwicklungsumgebung

Die unten mit * gekennzeichneten Tools sind auch auf der Praktikumshomepage verfügbar. Installationshinweise sind in den einzelnen Paketen enthalten. Die Verwendung von Eclipse (Schritte c) und d)) ist *optional*. Sie können zum Entwickeln von JavaCC-Code auch einen beliebigen anderen Editor verwenden.

a) Installieren Sie den Java SE 5.0 Development Kit (J2SE 5.0), der auf der Homepage `http://java.sun.com/` verfügbar ist. Falls Sie eine neuere Java-Version benutzen, müssen Sie sicher stellen, dass Ihr Java Source Code auch mit J2SE 5.0 fehlerfrei übersetzt und ausgeführt werden kann.

b) Installieren Sie den Java Compiler-Compiler *JavaCC 4.0**, welcher von der Homepage `https://javacc.dev.java.net/` heruntergeladen werden kann.

c) Installieren Sie *ANT 1.7.0**, welches von der Homepage `http://ant.apache.org/` heruntergeladen werden kann. Mit ANT können Sie die Kompilation von Java-Programmen plattformunabhängig machen und automatisieren. Ein Beispiel-ANT-Skript finden Sie auf der Praktikumshomepage.

d) Installieren Sie die Entwicklungsumgebung *Eclipse 3.2.1*. Die Software kann von der Eclipse Homepage `http://www.eclipse.org/` heruntergeladen werden.

e) Installieren Sie das JavaCC Plug-In (Version 1.5.6*) für Eclipse zur Unterstützung beim Editieren und Übersetzen von JavaCC-Code. Entpacken Sie die ZIP-Datei einfach im Stammverzeichnis Ihrer Eclipse-Installation (welche das `plugins` Verzeichnis enthält), und starten Sie Eclipse neu. Das Plugin kann von folgender URL heruntergeladen werden:

`http://sourceforge.net/projects/eclipse-javacc`

2. Erste Schritte mit JavaCC

a) Studieren Sie die Beispiele im Verzeichnis `example/SimpleExamples` Ihrer JavaCC-Installation. Ändern und kompilieren Sie die Beispiele in Ihrer Eclipse Entwicklungsumgebung. Verwenden Sie für die Kompilation und Ausführung ein ANT-Skript.

b) Lesen Sie die Dokumentation `doc/javaccgrm.html` Ihrer JavaCC-Installation, welche die JavaCC-Grammatik beschreibt.

3. Lexikalische Analyse (Scanner)

a) Ermitteln Sie die Token der Sprache $YAPL_4$.

b) Implementieren Sie mit Hilfe von JavaCC einen Scanner für die Sprache $YAPL_4$. Schreiben Sie eine Produktion, mit der Sie den Scanner testen können. Diese Produktion könnte z.B. so aussehen:

```
void Start() : { Token t; } {
  { do { t= getNextToken();
       System.out.println(t.toString());
    } while (t.kind != EOF);
  }
  <EOF>
}
```

c) Erlauben Sie die Existenz von Kommentaren in $YAPL_4$ Quelldateien durch eine entsprechende `SPECIAL_TOKEN` JavaCC Produktion. Kommentare werden (wie in C) von den Zeichenfolgen `/*` und `*/` eingeschlossen, eine Schachtelung von Kommentaren ist nicht erlaubt.

4. Ausdrücke in $YAPL_4$

a) Zeichnen Sie für den $YAPL_4$-Ausdruck

```
FALSE OR NOT 1 < 2 AND TRUE == 5 < 2 + 3 * 3
```

einen vereinfachten Ableitungsbaum, in dem Ketten (also Teilbäume, in denen jeder Knoten nur 1 Nachfolger besitzt) nur durch den Anfangs- und Endknoten dargestellt werden. Beginnen Sie die Ableitung bei `Expr`.

Beispiel: Die Kette `CondAndExpr - EqualExpr - ... - Literal - FALSE` wird durch `CondAndExpr - FALSE` dargestellt.

b) Ordnen Sie die Operatoren der $YAPL_4$ Sprache nach ihrem Vorrang (Präzedenz) in Ausdrücken (z.B. Punkt- vor Strichrechnung). Bei gleichrangigen binären Operatoren gilt Linksassoziativität, also zum Beispiel: $a - b + c = (a - b) + c$.

5. LL(n) Parsing mit JavaCC

a) Studieren Sie das *Lookahead* Mini-Tutorial von JavaCC, welches Sie unter `doc/lookahead.html` Ihrer JavaCC-Installation finden.

b) Schreiben Sie mit Hilfe von JavaCC einen Parser für folgende Sprache:

```
S = (A | "1") "*"
A = ("0" | "1") {"0" | "1"}
```

c) Implementieren Sie sämtliche Produktionen der $YAPL_4$ Grammatik in JavaCC, und identifizieren Sie alle Entscheidungspunkte (*choice points*) der Grammatik, die nicht mit einem Lookahead von 1 Token entschieden werden können. Fügen Sie geeignete `LOOKAHEAD` Direktiven in die betroffenen Produktionen ein. Wie lässt sich das *Dangling ELSE* Problem der $YAPL_4$ Grammatik lösen?

6. Syntaktische Analyse (Parser)

Implementieren Sie mit Hilfe von JavaCC einen Parser für die Sprache $YAPL_4$. Wenn die Eingabedatei ein syntaktisch korrektes $YAPL_4$ Programm darstellt, soll der Parser die Meldung

```
program <program> conforms to YAPL4 syntax.
```

ausgeben. Ist dies nicht der Fall, so soll eine Fehlermeldung der folgenden Form ausgegeben werden:

```
[lexical | parse] error in program <program> at line <l> column <c>:
    <error message>
```

Diese Form der Fehlermeldungen ist auch für alle folgenden Aufgaben einzuhalten. Bei Auftreten eines Fehlers darf der $YAPL_4$ Compiler die Ausführung sofort beenden. Fehler des von JavaCC generierten Parsers erzeugen eine `ParseException`, Fehler des Scanners einen

`TokenMgrError`. Lesen Sie dazu das *JavaCC Error Reporting and Recovery* Mini-Tutorial, das Sie unter `doc/errorrecovery.html` Ihrer JavaCC-Installation finden.

7. Gültigkeitsprüfung von Bezeichnern (Scoping)

a) Auf der Praktikumshomepage finden Sie Java-Schnittstellen für die Symboltabelle und Symbole (Bezeichner) der Sprache $YAPL_4$. Implementieren Sie diese Schnittstellen durch entsprechende Java-Klassen so weit, wie dies für die Gültigkeitsprüfung von Bezeichnern erforderlich ist. Beachten Sie die Kommentare und Implementierungshinweise in den Java-Schnittstellen. Die Java-Klassen des `yapl4.lib` Pakets sind selbst zu implementieren.

b) Implementieren Sie die Gültigkeitsprüfung von Bezeichnern eines $YAPL_4$ Programms. Variable und Prozeduren müssen deklariert werden, bevor sie verwendet werden können. Bezeichner sind nur innerhalb des Gültigkeitsbereichs (Produktionen `Block`, `Procedure`, `ProgramBlock`, `Program`) gültig, in dem sie deklariert wurden. Bezeichner müssen innerhalb eines Gültigkeitsbereichs eindeutig sein. Das Überladen von Prozeduren (unterschiedliche Parameterliste für denselben Prozedurnamen) ist daher nicht erlaubt. Bezeichner in inneren (verschachtelten) Gültigkeitsbereichen überdecken gleichnamige Bezeichner in äußeren Gültigkeitsbereichen. Rekursive Prozeduren werden unterstützt. Als Bezeichner für eine Arraydimension (siehe `ArrayDim`) sind nur Konstante erlaubt. Die Bezeichner am Beginn und Ende einer Prozedur bzw. des Programms müssen übereinstimmen. Der Programmbezeichner (Produktion `Program`) bildet einen eigenen Gültigkeitsbereich, der den Gültigkeitsbereich der Prozeduren und globalen Variablen umschließt.

Besteht das Eingabeprogramm die Gültigkeitsprüfung, soll der Compiler die Meldung

```
YAPL4 program <program> passed symbol validity test.
```

ausgeben. Die Erfolgsmeldung der Syntaxanalyse soll nicht ausgegeben werden.

Ihr Compiler muss folgende Fehler erkennen und die entsprechenden Fehlermeldungen ausgeben können:

- `END <name> does not match program <name>`
- `END <name> does not match procedure <name>`
- `symbol <name> already declared in current scope (as <kind>)`
- `identifier <name> not declared`
- `illegal use of <kind> <name>`
  Beispiele: Verwendung eines Prozedurbezeichners (ohne Klammern) in einem Ausdruck, Prozeduraufruf mit Variablenbezeichner, Zuweisung an Konstante, Deklaration einer Arraydimension durch Variablenbezeichner.

Dabei kann `<kind>` die folgenden Werte annehmen:
`program, procedure, variable, constant, reference parameter, constant reference parameter.` (Zu Prozedurparametern siehe Aufgabe 8.)

Es wird empfohlen, dass alle Fehlermeldungen des Compilers in einer Klasse `YAPLException` zusammengefasst und durchnummeriert werden, sodass eine Fehlermeldung durch Auslösen einer Exception mit der entsprechenden Fehlernummer (symbolische Konstante) und den nötigen Parametern (`Symbol` und/oder `Token` Objekt) generiert werden kann.

c) Erweitern Sie die $YAPL_4$ Grammatik und den Compiler um Produktionen für die Deklaration der vordefinierten Prozeduren (siehe Seite 3). Lassen Sie den Compiler vor dem Einlesen der Programmdatei die Deklarationen der vordefinierten Prozeduren einlesen (suchen Sie nach der `reInit()` Methode in der JavaCC API Dokumentation). Der einfacheren Implementierung wegen sollen die vordefinierten Prozeduren einen eigenen, äußersten Gültigkeitsbereich bilden, der das Programm umschließt. Benutzerdefinierte Variablen und Prozeduren können daher vordefinierte Prozeduren überdecken. Verwenden Sie die auf der Praktikumshomepage verfügbaren $YAPL_4$ Quelldateien, um den Compiler zu testen.

8. Typüberprüfung (Type Checking)

Der $YAPL_4$ Compiler soll ein Eingabeprogramm auf die korrekte Verwendung der $YAPL_4$ Datentypen überprüfen. Dafür gelten folgende semantische Regeln:

- *Typkompatibilität:*
  - Die Typen `INTEGER` und `BOOLEAN` sind untereinander sowie mit Arraytypen nicht kompatibel.
  - Das `CONST` Attribut beeinflusst die Typkompatibilität nicht.
  - Arraytypen sind kompatibel, wenn sie die gleiche Länge haben und die Elementtypen kompatibel sind.
  - Es findet keine Typkonversion statt.
- *Deklarationen:*
  - Der Bezeichner einer Arraydimension muss vom Typ `INTEGER` sein.
  - Die Elementanzahl einer Arraydeklaration muss größer als 0 sein.
  - Konstante (`ConstDecl`) erhalten den Typ des Literals, das bei der Deklaration zugewiesen wird: `INTEGER` oder `BOOLEAN`.
  - Primitive Parametertypen (`INTEGER, BOOLEAN`) einer Prozedur werden *by value* übergeben, Arraytypen *by reference*.
  - Das `CONST` Attribut eines formalen Parameters ist nur für Arraytypen (call by reference) zulässig und bedeutet, dass das referenzierte Objekt (Array) durch die Prozedur nicht verändert werden darf.
- *Zuweisungen:*
  - Zuweisungen sind nur an L-Werte der Typen `INTEGER` und `BOOLEAN` zulässig.
  - Die Typen von L- und R-Wert müssen kompatibel sein.
  - Zuweisungen an konstante L-Werte (`CONST`) sind nicht zulässig.
- *Ausdrücke:*
  - Die Typen der beiden Operanden eines binären Operators müssen kompatibel sein.
  - Die Operatoren `*, /, +, -, <` sind nur für `INTEGER` Operanden zulässig.
  - Der Operator `==` ist nur für `INTEGER` oder `BOOLEAN` Operanden zulässig.
  - Die Operatoren `NOT, AND, OR` sind nur für `BOOLEAN` Operanden zulässig.
  - Vergleichs- und Gleichheitsausdrücke (`RelExpr, EqualExpr`) haben den Ergebnistyp `BOOLEAN`.

  - Eine Prozedur ohne Rückgabetyp darf nicht in einem Ausdruck (`PrimaryExpr`) aufgerufen werden.

- *Prozeduraufrufe:*
  - Die Argumenttypen eines Prozeduraufrufes müssen kompatibel zu den Parametertypen der Prozedurdeklaration sein.
  - Ein konstanter Argumenttyp darf nicht einem nicht konstanten (aber kompatiblen) Referenzparametertyp übergeben werden.
- *Anweisungen:*
  - Bedingungen (`Expr`) in `IF` und `WHILE` Anweisungen müssen den Typ `BOOLEAN` haben.
  - Prozeduren mit Rückgabetyp (Funktionen) müssen eine `RETURN` Anweisung mit einem zum Rückgabetyp kompatiblen Ausdruck enthalten. (Es wird allerdings nicht geprüft, ob in jedem Ausführungspfad – etwa bei `IF` Anweisungen – eine `RETURN` Anweisung enthalten ist.)
  - Prozeduren ohne Rückgabetyp sowie das Hauptprogramm dürfen keine `RETURN` Anweisung mit Rückgabewert enthalten.

Bei erfolgreicher Typüberprüfung des Eingabeprogramms soll der Compiler die Meldung

```
YAPL4 program <program> passed type check.
```

ausgeben. Der Compiler soll folgende Fehler erkennen und die entsprechenden Fehlermeldungen produzieren können:

- `invalid type or value of array dimension`
  Nur bei Deklarationen.
- `constant call-by-value parameter <name>`
- `illegal operand type for unary operator <op>`
- `illegal operand types for binary operator <op>`
- `expression before '[' is not an array type`
- `array selector is not an integer type`
- `non-integer operand types for relational operator <op>`
- `illegal operand types for equational operator <op>`
- `type mismatch in assignment`
- `illegal lvalue type in assignment`
- `constant argument passed to non-constant reference parameter #<n> of procedure <name>`
  `<n>` ist die Argumentnummer; das erste Argument hat die Nummer 1.
- `argument #<n> not applicable to procedure <name>`
  Argumenttyp des Prozeduraufrufs ist nicht mit Parametertyp der Prozedurdefinition kompatibel, oder das Argument ist überzählig.
- `too few arguments for procedure <name>`
- `using procedure <name> (not a function) in expression`

- `returning invalid type from function <name>`
  In Funktionen bei `RETURN` Anweisungen, deren Rückgabewert (`Expr`) nicht vorhanden oder nicht kompatibel zum Rückgabetyp der Funktion ist.
- `missing RETURN statement in function <name>`
- `illegal return value in procedure <name> (not a function)`
- `illegal return value in main program`
- `condition is not a boolean expression`
  Bei `IF` und `WHILE` Anweisungen.

Es wird folgende Vorgangsweise bei der Implementierung vorgeschlagen:

a) Erzeugen Sie entsprechende Java-Objekte der $YAPL_4$ Datentypen beim Parsen der Deklarationen, und speichern Sie diese mit den Bezeichnern in der Symboltabelle. Bei Konstantendeklarationen ist zusätzlich auch der Wert der Konstanten in der Symboltabelle bzw. beim Datentyp zu speichern. Implementieren Sie zudem die Typüberprüfung für Deklarationen (inkl. Prozeduren).

b) Die Typüberprüfung von Ausdrücken kann während des Parsens ("on-the-fly") mit Hilfe von Grammatikattributen erfolgen, die von Produktion zu Produktion weitergereicht werden. Sie finden dazu auf der Praktikumshomepage eine Java-Schnittstelle `Attrib`, welche Zugriff auf die Attribute eines Operanden bzw. des Ergebnisses eines Ausdrucks bietet. Implementieren Sie die `Attrib` Schnittstelle durch eine entsprechende Java-Klasse so weit, wie es für die Typüberprüfung nötig ist.

c) Auf der Praktikumshomepage finden Sie zudem eine Java-Schnittstelle `CodeGen` für die Codegenerierung, welche bereits für die Typüberprüfung hilfreich sein kann. Implementieren Sie die Schnittstelle durch eine entsprechende Java-Klasse so weit, wie es für die Typüberprüfung von Ausdrücken nötig ist. Konstante Ausdrücke dürfen bei der Codegenerierung wie nicht-konstante Ausdrücke behandelt werden, müssen also *nicht* vom Compiler ausgewertet werden. Bei Verletzung der oben angeführten semantischen Regeln für $YAPL_4$ soll eine `YAPLException` geworfen werden (vgl. Aufgabe 7).

d) Implementieren Sie die Typüberprüfung für Ausdrücke (ohne Prozeduraufrufe), Zuweisungen, Prozeduraufrufe und Anweisungen (in dieser Reihenfolge). Verwenden Sie die auf der Praktikumshomepage verfügbaren $YAPL_4$ Quelldateien, um den Compiler zu testen.

9. Codegenerierung

Der $YAPL_4$-Compiler soll MIPS-Assemblercode – bekannt aus der Lehrveranstaltung Rechnerorganisation – generieren, der durch den SPIM Simulator ausführbar sein sollte. SPIM-Software und -Dokumentation finden Sie auf der Praktikumshomepage. Generell wird gefordert, dass der zu erzeugende MIPS-Code lediglich *korrekt*, aber *nicht optimiert* sein muss.

Für die Implementierung wird folgende Reihenfolge vorgeschlagen:

a) **Entwurf:** Vervollständigen Sie den Entwurf Ihres Compilers für die Codegenerierung. Als Hilfestellung finden Sie auf der Praktikumshomepage die Java-Schnittstellen `CodeGen` und `TargetArch`, die nützliche Methoden für die Codegenerierung auf zwei Abstraktionsebenen enthalten: die `CodeGen` Methoden stellen im Wesentlichen 3-Adress-

Code-Operationen dar (obwohl kein Zwischencode erzeugt wird) und sind unabhängig von der Zielsprache; sie rufen die `TargetArch` Methoden auf, um den Code in der Zielsprache zu erzeugen. Ihr $YAPL_4$ Parser kann somit die `CodeGen` Methoden aufrufen, um den MIPS Assemblercode *on-thy-fly*, also nur in einem Durchgang, zu erzeugen. Die Java-Schnittstellen sollten von Ihnen durch entsprechende Java-Klassen implementiert werden; die `TargetArch` Implementierung sollte MIPS Assemblercode erzeugen und könnte etwa `MipsArch` genannt werden.

Überlegen Sie, welche `CodeGen` und `TargetArch` Methoden jeweils aufgerufen werden müssen, um den Assemblercode der folgenden $YAPL_4$ Produktionen zu erzeugen: `VarDecl, Assignment, AddExpr, Procedure, ProcedureCall, IfStatement`. Denken Sie auch an die Erzeugung von Sprungmarken (Labels) und an die Verwaltung der MIPS-Register.

Erweitern Sie die Kommandozeilenschnittstelle Ihres Compilers, sodass der Dateiname für den erzeugten MIPS-Assembler-Code als Kommandozeilenargument übergeben werden kann. Definieren Sie geeignete Konstruktoren für die Java-Klassen, welche die `CodeGen` und `TargetArch` Schnittstellen implementieren sollen. Die `TargetArch` Implementierung sollte möglichst direkt in die Ausgabedatei schreiben (und nicht etwa ein `String` Objekt zurückliefern).

b) **Prozedurdefinitionen ohne Parameter:** Implementieren Sie die Codegenerierung für Prozedurdefinitionen ohne formale Parameter sowie für das Hauptprogramm (Produktion `ProgramBlock`), welches im erzeugten MIPS-Code durch die Sprungmarke `main` (Programmeinstiegspunkt) gekennzeichnet sein muss. Verwenden Sie folgende Struktur des *Activation Record* für Prozeduren (und für das Hauptprogramm):

```
(Stack wächst nach unten)
--- Stackframe der aufrufenden Prozedur:
    --- gesicherte Register (falls benötigt):
        $v0, $v1
        $a0 - $a3
        $t0 - $t9
        $s0 - $s7 (*)
    --- Prozedurargumente:
        ...
        arg 2
        arg 1
        arg 0
--- Stackframe der aufgerufenen Prozedur:
    --- gesicherte Register:
$fp ->  $fp
        $ra
    --- lokale Variable:
        ...
$sp ->  (nächste freie Adresse am Stack - top of stack)
```

(*) *Anmerkung:* Der einfacheren Implementierung wegen dürfen die s-Register als caller-saved Register behandelt werden – obwohl dies der MIPS-Prozeduraufrufskonvention widerspricht.

Die `$fp` und `$sp` Register sollten zu Beginn einer Prozedur so initialisiert werden, dass

sie auf den Anfang bzw. das Ende des Stackframes der Prozedur zeigen. Sämtliche Prozedurargumente sollten am Stack übergeben werden, der Rückgabewert ggf. in `$v0`. Die aufrufende Prozedur sollte vor dem Aufruf nur jene Register sichern, deren Werte sie nach dem Prozeduraufruf wieder benötigt.

c) **Vordefinierte Prozeduren:** Auf der Praktikumshomepage finden Sie den MIPS-Assemblercode für die Prozedur `write`, welche für die Codegenerierung der `WRITE` Anweisung benötigt wird. Schreiben Sie analog den MIPS-Code für die vordefinierten Prozeduren der $YAPL_4$ Sprache und fügen Sie diesen in das `.text` Segment des erzeugten MIPS-Codes ein. Verwenden Sie für die Ein-/Ausgabeoperationen die im SPIM Simulator verfügbaren System Calls. Beachten Sie die oben definierte Struktur des Activation Record.

d) **Prozeduraufrufe und WriteStatement:** Implementieren Sie die Codegenerierung für Prozeduraufrufe mit call-by-value Parametern und Rückgabewert sowie für die Produktion `WriteStatement`. Testen Sie Ihren Compiler mit einem $YAPL_4$ Programm, das konstante Werte ausgeben kann. $YAPL_4$ Quelldateien sind auf der Praktikumshomepage verfügbar.

e) **Deklaration globaler Variablen:** Alle globalen Variablen müssen im Datensegment (`.data`) des erzeugten MIPS-Programms abgelegt werden. Der Zugriff auf die Variablen sollte mittels Offset erfolgen, das beim jeweiligen Symbol in der Symboltabelle gespeichert wird. Der Einfachheit halber belegen `INTEGER` und `BOOLEAN` Datentypen jeweils 4 Bytes. Der erzeugte MIPS-Code und die Offset-Werte könnten beispielsweise so aussehen:

```
YAPL4-Code:                    MIPS-Code:              Offsets:
-----------                    ----------              --------
PROGRAM Test                   .data
VAR                            globals:
  x: INTEGER;                    .space 4  # x         x:  0
  a: BOOLEAN;                    .space 4  # a         a:  4
  y: ARRAY 10 OF INTEGER;        .space 40 # y         y:  8
  z: INTEGER;                    .space 4  # z         z: 48
...
```

f) **Einfache Zuweisungen:** Implementieren Sie die Codegenerierung für die Produktion `Assignment` soweit, dass konstante Werte an globale Variable des Typs `INTEGER` oder `BOOLEAN` zugewiesen werden können. Testen Sie Ihren Compiler mit einem $YAPL_4$ Programm, das Werte von globalen Variablen ausgibt.

g) **Ausdrücke:** Implementieren Sie die Codegenerierung von Ausdrücken, wobei darauf zu achten ist, dass für alle Operationen eines Ausdrucks auch entsprechender MIPS-Code erzeugt wird (der Ausdruck muss *zur Laufzeit* ausgewertet werden!); dies gilt auch für konstante Ausdrücke (vgl. Aufgabe 8 c)). Bei Booleschen Ausdrücken können Sie zwischen einer numerischen Darstellung und einer Kontrollflussimplementierung wählen. Optimierungen wie *lazy evaluation* sind nicht gefordert. Erzeugen Sie für Selektoren von Arrays zunächst noch keinen Code. Testen Sie Ihren Compiler durch $YAPL_4$ Programme, die das Ergebnis von Ausdrücken ausgeben bzw. an globale Variable zuweisen.

h) **Arrays:** Implementieren Sie die Codegenerierung für Selektoren von Arrays. Achten Sie darauf, dass Indexausdrücke und Adressen von Arrayelementen *zur Laufzeit* berechnet und in Ausdrücken und L-Werten von Zuweisungen richtig verwendet werden.

i) **Kontrollanweisungen:** Implementieren Sie die Codegenerierung für `IfStatement` und `WhileStatement`. Optimierungen wie etwa die Eliminierung unnötiger Sprünge sind *nicht* gefordert.

j) **Lokale Variable:** Variable, die innerhalb eines Blocks (Produktion `Block`) deklariert werden, müssen am Stack gespeichert werden (siehe Activation Record oben). Sie sollten mittels Offset relativ zum Framepointer (`$fp`) adressiert werden. Achten Sie darauf, dass auch die Adressen von Arrayelementen am Stack richtig berechnet werden. Testen Sie Ihren Compiler mit einem $YAPL_4$ Programm, das lokale Variable im Hauptprogramm deklariert und verwendet.

k) **Prozeduren:** Vervollständigen Sie die Codegenerierung für Prozedurdefinitionen und Prozeduraufrufe. Es wird folgende Implementierungsreihenfolge empfohlen:

- *Call-by-value Parameter*.
- *Call-by-reference Parameter*. Dies sind in $YAPL_4$ stets Arrays. Die (absolute) Startadresse des Arrays wird zur Laufzeit als Prozedurargument übergeben.
- *ReturnStatement*. Der erzeugte Assemblercode sollte nicht direkt zur aufrufenden Prozedur zurückkehren, sondern zum Prozedurepilog (ans Ende der Prozedur, welche die `RETURN` Anweisung enthält) springen, wo der Stackframe wieder freigegeben wird. Ein allfälliger Ausdruck für den Rückgabewert muss zuvor ausgewertet werden.

Testen Sie Ihren Compiler abschließend durch rekursive Prozeduren sowie durch eine QuickSort-Implementierung in $YAPL_4$ (auf der Praktikumshomepage im Rahmen der $YAPL_4$ Testdateien verfügbar).