Dokumentation des SASL-Compilers

Birgit Berchtold & Verena Hafner
Informatik IV
Software-Grundpraktikum
Sommersemester 1997
Universität Konstanz

1 Einleitung

Das Ziel unseres Projektes ist es, für die funktionale Programmiersprache SASL (St. Andrews Static Language) einen Compiler in der objektorientierten Programmiersprache Eiffel (Tower Eiffel 3.0) zu realisieren. Die Vorgehensweise dabei war, vorgegebene SASL-Files in sogenannte Parsebäume zu übersetzen, die dann nach Eliminierung von eventuellen Variablen solange reduziert werden, bis das Ergebnis des Programms erhalten ist.

2 Nutzereinführung

Für die Anwendung des Compilers `bvs` geben wir an dieser Stelle eine kurze Nutzereinführung an.

2.1 Die man-page

BVS(1)

NAME
  bvs – compiles sasl sourcecodes

SYNOPSIS
  bvs <sasl-file>
  bvs –version, –help, –debug

WARNING
  The information in this man page is an extract from the full documentation of the sasl compiler, and is limited to the meaning of the options.

This man-page is not kept up to date except when volunteers want to maintain it.

DESCRIPTION

bvs compiles a sasl-sourcecode file and displays the result to the standard output.

OPTIONS

–version
: Print version information on standard output.

–help
: Display usage message on standard output.

–debug <sasl-file>
: Print parsetree after parsing on standard output.

2.2 Prelude

Ein Prelude (Funktionenbibliothek) kann vor den SASL-Sourcecode gehängt werden. Eine Integration in unseren Compiler ist in Planung.

2.3 Interpretation der Ausgabe und Fehlermeldung

Nach Beendigung des Compilationsprozesses wird das Ergebnis des Programms, eine Konstante (also ein Integer, ein String ein Boolean oder eine Liste), auf dem Bildschirm ausgegeben. Wird beim Aufruf der Compilation zusätzlich die Option *debug* gewählt, so wird vor der Ausgabe des Endergebnisses noch der Parsebaum, der das SASL-Programm darstellt, und der compilierte und abstrahierte Parsebaum (vergleiche dazu Abschnitt 3.7) ausgegeben. An oberster Stelle befindet sich jeweils der Wurzelknoten des Baumes. Die beiden Nachfolger eines jeden Knotens werden in entsprechender Tiefe eingerückt dargestellt.
Als Beispiel betrachte man folgendes SASL-Programm:

```
def f=42.
f
```

Der dazugehöhrige Parsebaum hat die folgende Gestalt:

```
              Pdot
              /  \
          Pdef    Pvar
          /  \
   Pvar : f  Pnum : 42
```

Das Ausgabefeature stellt dann den Baum folgendermaßen dar:

```
Pdot
___Pdef
______Pvar : f
______Pnum : 42
___Pvar : f
```

bvs erzeugt bei folgenden Situationen Fehlermeldungen und bricht mit Exitcode 1 ab:

- Bei falschem Aufruf von bvs wird eine usage ausgegeben. Die usage hat folgende Gestalt:

  ```
  usage: bvs [--debug] <SASL-File>
         bvs {--help,--version}
  ```

- Wenn das angegebene SASL-Sourcefile nicht existiert, bricht der Lexer ab mit der Meldung

  ```
  Das File gibt's nicht....
  ```

- Der Parser kann grobe Fehler in der Syntax von SASL erkennen und bricht dann ab. Zum Beispiel löst das SASL-Programm

  ```
  def f=+.
  f
  ```

  die Meldung

  ```
  Fehler !!!
  Erwartet:32
  im SASL:34
  ```

  aus.
  Inhaltliche Fehler jedoch werden nicht erkannt, wie z. B. das Verwenden einer vorher nicht deklarierten Variablen. Mehrfach definierte Variablen werden ebenfalls nicht erkannt.

  Beispiel 1):

  SASL-Programm:

  ```
  def x=42
  def x=1.
  x
  ```

Parsebaum:

```
                  Pdot
                 /    \
      Pdef,     Pdef      Pvar
     /    \    /   \
Pvar : x Pnum : 42 Pvar : x Pnum : 1
```

- Der Compiler bricht ab, falls Variablen auftreten, die nicht definiert wurden.

  Beispiel 2):

  ```
  def f x = x.
  x
  ```

  Dieses SASL-Programm führt zu folgender Fehlermeldung:

  `Keine Ref für x gefunden!`

  Werden dagegen Variablen mehrfach definiert, so wird vom Compiler nur die erste Definition benutzt. Das SASL-Programm aus Beispiel 1) oben gibt nach der Compilation und Abstraktion "nur" noch folgenden Baum aus:

  ```
  Pnum : 42
  ```

3 Funktionsweise des Compilers

Grundlegend besteht unser Compiler aus den vier Komponenten 'Lexer', 'Parser', 'Compiler' und 'Reducer'. Diese Komponenten werden von der Hauptklasse `bvs` angesprochen, die die Compilation steuert.

3.1 Klassenstruktur

Um die Klassenstruktur sowie die Beziehung der Instanzen der einzelnen Objekte untereinander möglichst intuitiv darzulegen, werden wir uns an die von James Rumbaugh in seinem Artikel 'The life of an object: How the object model changes during development' vorgeschlagene *Object Modeling Technique* (kurz: OMT) anlehnen. Das unten angeführte Klassendiagramm (Abbildung 1) ist in dieser OMT-Notation dargestellt.

Eine Klasse wird dabei durch einen Kasten angedeutet. Der Name der Klasse ist im oberen Feld des Kastens durch Fettdruck hervorgehoben. Die öffentlichen

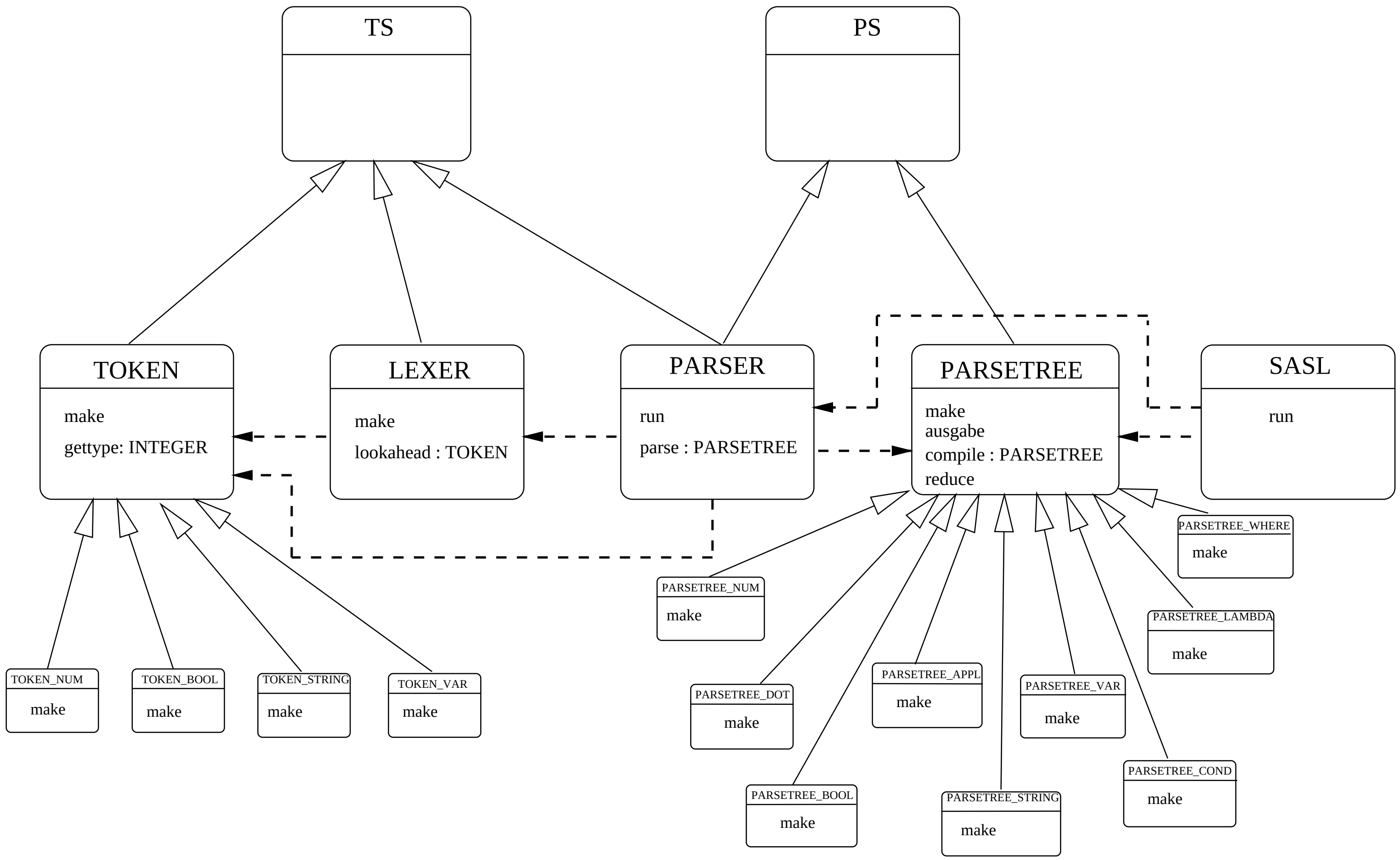

Abbildung 1: Klassendiagramm

features der Klasse folgen im unteren Teil des Kastens. Zusätzlich zu den feature-Namen sind hier auch Return-Typen angegeben. Ist kein Return-Typ angegeben, so gibt das feature keinerlei Rückgabewerte zurück. Die OMT-Notation für Klassenvererbung ist eine Linie mit einem Dreieck, die eine Unterklasse mit der jeweiligen Elternklasse verbindet. Die Spitze des Dreiecks zeigt hierbei auf die Elternklasse. Um anzuzeigen, welche Klasse Objekte von anderen Klassen erzeugt, wird eine gestrichelte Linie mit einem Pfeil verwendet. Der Pfeil zeigt hierbei auf die Klasse, von der das betreffende Objekt erzeugt wurde.

3.2 Lexer

Der Lexer ist für das Einlesen des zu compilierenden SASL-Codes und dessen syntaktischer Umwandlung in sogenannte Tokens zuständig. Der Lexer liefert auf Anfrage des Parsers das jeweils nächste SASL-Code Segment in Form eines Tokens an den Parser aus.

3.3 Parser

Das Ziel des Parsers ist der Aufbau eines Parsetrees, welcher komplett an den Compiler übergeben werden kann. Hierzu prüft der Parser jedes vom Lexer erhaltene Token mit eventuellem Argument auf Verträglichkeit mit der SASL-Syntax und baut den Inhalt des Tokens in Form eines Parsetree-Knotens, bei Syntaxverträglichkeit der SASL-Grammatik entsprechend in den Parsetree ein. Wichtig hierbei ist, da nicht das Token selbst, sondern der durch die Grammatik dem Token zugeordnete Parsetree eingebaut wird. Der gesamte Parsetree besteht also in jedem seiner Knoten aus einem korrespondierend zu Token und Grammatik minimalen Parsetree.

3.4 Compiler

Aufgabe des Compilers ist es, Variablen aus dem Parsetree zu entfernen, damit dieser anschließend von der Reduktionsmaschine bearbeitet werden kann. Die Objektorientierung von Eiffel bietet die Möglichkeit an, eigene `compile`-features für die einzelnen Parsetree-Klassen zu erstellen. Dazu implementiert man das "Standard-`compile`-feature" in der Hauptklasse und spezialisiert gegebenenfalls in den einzelnen Unterklassen.

3.5 Reducer

Wie der Name schon sagt, reduziert der Reducer den vom Compiler vorbereiteten Baum, bis dieser nur noch aus einem einzelnen Knoten besteht. Dieser Knoten enthält das Ergebnis des Programms, welches dann ausgegeben werden kann. Die Berechnung dieses Ergebnisses erfolgt durch die in Eiffel eingebauten Funktionen.

Wie beim Compiler wurden die features in die einzelnen Parsetree-Klassen eingebaut. Dadurch erspart man sich die Abfrage, welcher Parsetree-Klasse der gerade zu reduzierende Knoten zuzuordnen ist. Das Laufzeitsystem von Eiffel erkennt die Parsetree-Klasse und wendet dann automatisch das passende `reduce`-feature an. Diese Vorgehensweise wird auch *late-binding* genannt.

3.6 Die Hauptklasse `bvs`

- Die Hauptklasse `bvs` (**b**irgit**v**erena**s**asl :-)) erstellt eine Instanz p der Klasse `Parser`. Dabei wird der Name des SASL-Files, welcher von dem Benutzer als Argument mitgegeben wird, übergeben.
- Das creation-feature von p wird aufgerufen. Dieses erzeugt eine Instanz l der Klasse `Lexer` und ruft deren creation-feature mit dem SASL-Filenamen als Argument auf.
- l öffnet das SASL-File, bzw. erzeugt eine Fehlermeldung, falls es nicht existiert.
- `bvs` fordert von p den Parsebaum des SASL-Files durch `p.parse` an.
- p holt sich jeweils von l durch `l.lookahead` das nächste Token, prüft den SASL-Quelltext auf seine Korrektheit und erstellt dabei den Parsebaum.
- Durch `baum.baumcompile` erhält `bvs` den compilierten und abstrahierten Parsebaum `fertig_baum`.
- `bvs` erhält den reduzierten Baum `redbaum` durch `fertig_baum.startreduce`.
- Das Ergebnis wird dann ausgegeben.

Vergleiche dazu auch das Interaktionsdiagramm in Abbildung 2.

3.7 Implementation des Designs

In unserer Implementation bilden der Lexer sowie der Parser zwei eigenständige Klassen.

3.7.1 Implementation des Lexers

Der Lexer ist dafür verantwortlich, das SASL-File zu öffnen, und gibt, falls es nicht existiert, eine Fehlermeldung aus und bricht mit Exitcode 1 ab. Dies geschieht in dem creation-feature `make`. In dem öffentlichen feature `lookahead`, welche vom Parser aufgerufen wird, um ein Token anzufordern, liest der Lexer jeweils so lange Zeichen aus dem File, bis er ein vollständiges Token hat. Dabei unterscheidet

er zwischen Integern (unsere SASL-Implementation läßt keine Real-Zahlen zu), Strings, welche in Anführungszeichen stehen müssen, Schlüsselwörtern, Booleans und Variablen. Die Schlüsselwörter sind in einer Hash-Tabelle gespeichert und können dort verglichen werden. Dabei wird darauf geachtet, daß es längere Tokens geben kann, die "versehentlich" bereits ein kürzeres Token enthalten können wie zum Beispiel '<' und '<='. Ein vollständiges Token wird dadurch erkannt, daß der Lexer auf ein Whitespace, ein Return, ein Tab oder auf das File-Ende stößt, oder das Token befindet sich in der Hash-Tabelle.
Falls ein Token weder Integer, Boolean, String noch Schlüsselwort ist und aus den Zeichen ['a',..,'Z','0',..,'9','_'] besteht, so wird es als Variable aufgefaßt. Andernfalls wird eine Fehlermeldung erzeugt, und das Programm wird mit Exitstatus 1 abgebrochen.
Wenn das Ende des Files erreicht ist, gibt der Lexer ein spezielles Token 'Teof' zurück, damit der Parser weiß , daß er keine weiteren Tokens anfordern kann, da der SASL-Code hier zu Ende ist.

3.7.2 Implementation des Parsers

Der Parser hat zwei öffentliche features: Das creation-feature `run` erzeugt eine Instanz l der Klasse `Lexer` und führt dessen creation-feature `make` aus, welche das SASL-File zu öffnen versucht. Bereits hier kann abgebrochen werden, falls das File nicht existiert. Andernfalls holt sich der Parser über `l.lookahead` das erste Token und speichert es in der Variablen `nexttoken` ab.
Das feature `parse` startet nun den eigentlichen Parse-Vorgang. Die SASL-Grammatik wurde folgendermaßen implementiert: Die in *EBNF* (erweiterte Backus-Naur-Form) notierte Grammatik wurde durch Elimination von Linksrekursion und Linksfaktorisierung in eine für den Parser brauchbare Form transformiert.
Linksrekursion bedeutet, daß eine Grammatikregel der Form

$$\begin{array}{ccl} A & \longrightarrow & A\alpha \\ & | & \beta \end{array}$$

äquivalent umgeformt wird in die Form

$$\begin{array}{ccl} A & \longrightarrow & \beta A' \\ A' & \longrightarrow & \alpha A' \\ & | & \varepsilon, \end{array}$$

ohne ihre Bedeutung zu verändern. α steht hier für eine beliebige Folge von (Nicht-)Terminalen, β für eine Folge von (Nicht-)Terminalen, die nicht mit A beginnt; ε repräsentiert das leere Wort.
Grammatikregeln, der Form

$$\begin{array}{ccl} A & \longrightarrow & \alpha\beta_1 \\ & | & \alpha\beta_2 \end{array}$$

werden in die Form

$$\begin{array}{rcl} A & \longrightarrow & \alpha A' \\ A' & \longrightarrow & \beta_1 \\ & | & \beta_2 \end{array}$$

transformiert. Diese Transformation wird Links-Faktorisierung genannt.
Für jede diese nun neuen Grammatikvorschriften gibt es im Parser eine eigene private Funktion, welche jeweils einen Parsetree zurück gibt (recursive descent parsing).
`parse` muß nun die Funktion `system` starten; die Grammatik wird dann durch (rekursive) Aufrufe verglichen und der Parsebaum erstellt. Dabei können die einzelnen Grammatik-Funktionen auf die ebenfalls private Prozedur `match` zugreifen, welche ein von der Grammatik erwartetes Token mit dem aktuellen vom Lexer geholten Token `nexttoken` vergleicht und bei Übereinstimmung ein neues Token vom Lexer holt. Falls die Tokens nicht übereinstimmen, kann die Syntax des SASL-Sourcecodes nicht stimmen, und der Parse-Vorgang wird mit einer Fehlermeldung abgebrochen.
Jede der Grammatik-Funktionen ist dafür zuständig, einen Teil-Parsetree, welcher Teil des gesamten Parsetrees ist, aufzubauen. Der Aufbau dieser Parsetrees ist durch diese Funktionen vordefiniert und hängt natürlich zusätzlich von den vom Lexer erhaltenen Tokens ab.

3.7.3 Implementation des Compilers

SASL-Programme bestehen aus eventuellen (globalen) Variablen-Definitionen und aus dem zu berechnenden Ausdruck. Treten (globale) Variablen-Definitionen auf, durchsucht der Compiler den zu berechnenden Ausdruck nach Variablen. Sobald eine Variable gefunden wurde, wird sie durch die zugehörige Definition ersetzt. Dazu wird dem feature `compile` die Liste der Definitionen als Array übergeben. Die eigentliche Ersetzung wird durch das feature `fixup` ausgeführt, welches in diesem Array die der Variablen zugehörige Definition sucht und übergibt. Da es nun auch Funktionen gibt, die auf ihr eigenes Argument zugreifen, muß in solchen Funktionen-Definitionen eine sogenannte Abstraktion durchgeführt werden. Dieses erledigt das feature `abstract`. Dazu werden sogenannte Combinator-Knoten eingeführt, die das Argument der Funktion aus der Funktionen-Definition entfernen, ohne dabei die Bedeutung des Programms zu verändern.
Als Beispiel für einen compilierten und abstrahierten Parsebaum betrachte man das folgende SASL-Programm:

```
def f x = x.
f 42
```

Der zugehörige Parsebaum

```
                Pdot
              /      \
         Pdef          Pappl
        /    \        /     \
Pvar : f  Plambda  Pvar : f  Pnum : 42
            /     \
    Pvar : x    Pvar : x
```

hat dann nach der Compilation und Abstraktion folgende Gestalt:

```
                Pappl
               /    \
            PI     Pnum : 42
```

3.7.4 Implementation des Reducers

Die Implementation des Reducers ist aus zeitlichen Gründen leider noch nicht vollendet. Sie wird so bald wie möglich nachgereicht. Die Idee dabei ist aber folgende:
Der Reducer geht so vor, daß er zuerst immer die linken Nachfolger eines jeden Knotens untersucht. Solange diese Funktionsanwendungsknoten sind, werden diese auf einem Stack abgelegt, bis es auf einen Funktionsknoten trifft. Die Ablage auf einen Stack ist deshalb wichtig, damit die Vorgänger des Funktionsknotens nicht verloren gehen. In vielen Fällen enthalten diese nämlich Verweise auf die Argumente der Funktion. Der Funktionsknoten kann ein vom Compiler geschaffener Combinator-Knoten oder ein einer eingebauten SASL-Funktion zugeordneter Knoten sein. Je nach Funktionsvorschrift tranformiert der Reducer den Baum in einen Graphen. Daß jeweils die richtige Transformationsregel angewandt wird, erreicht man dadurch, daß in den Parsetree-Klassen eines jeden Funktionsknotens das passende feature `reduce` implementiert wird. Die eigentliche Berechnung wird durch die in Eiffel eingebauten Funktionen ausgeführt. Durch sukzessives Transformieren wird schließlich das Ziel erreicht, nämlich, daß nur noch ein einziger Knoten übrig bleibt.

3.8 Interaktionsdiagramm

Das Interaktionsdiagramm (Abbildung 2) soll den zeitlichen Ablauf der gesamten Compilation eines Programmes grafisch darstellen.

4 Einschränkungen des Compilers

SASL-Programme bestehen aus eventuellen globalen Definitionen und dem zu berechnenden Ausdruck. Dieser kann wiederum lokale Definitionen enthalten, die nur innerhalb dieses Ausdrucks bekannt sind. Solche lokalen Definitionen werden vom Compiler ebenfalls (wie bei der Abstraktion) mit speziellen Combinator-Knoten dargestellt. Unsere Version des Compilers läßt allerdings noch keine solchen lokalen Definitionen zu. Zwar werden SASL-Programme, die lokale Definitionen enthalten, im Augenblick vom Parser erkannt und verwaltet, der Compiler jedoch kann diese nicht compilieren. Da man aber SASL-Programme, die lokale Definitionen enthalten, umformulieren kann in Programme, die nur mit globalen Definitionen auskommen, verliert man nicht an Ausdruckskraft.

Eine weitere Einschränkung des Compilers ist, daß keine Real-Zahlen zugelassen sind (dies ist jedoch in der SASL-Grammatik auch gar nicht vorgesehen); der Compiler verwaltet nur Integer-Werte. Bei Division von Integern wird dann eine ganzzahlige Division durchgeführt.

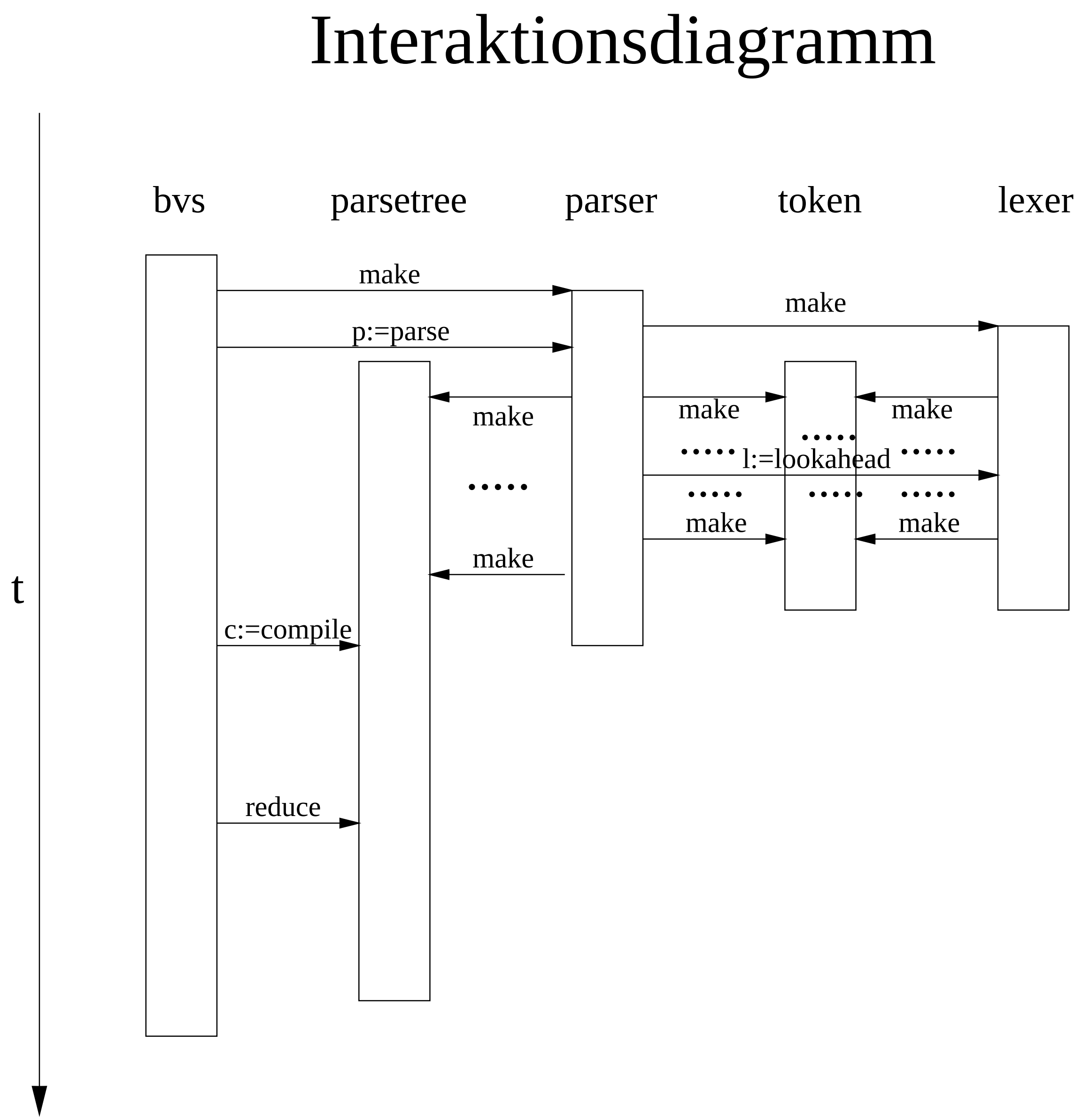

Abbildung 2: Interaktionsdiagramm