Das Wesen des Teils.

Philip Wagner

Dªs Wesen des Teils

Bachelor Report über
Objekt Orientierten Ontologismus

Hochschule für Künste Bremen
Wintersemester 2012/2013

Erstprüferin Prof. Dr. Andrea Sick
Zweitprüfer Prof. Dennis Paul

Raufasertapete, Macbook Pro Retina 15”, Papier, Tinte, Schweiß, Angst, Orangensaft, Hochschule für Künste Bremen, InDesign, Javascript, Storytelling, Mimikry, Land of Lisp, OOO, Hyperobjekt, Carl Sagan, Minion Pro, toy.cpp, docco, Videospiele, Martin Heidegger, Calvinball, Notational Velocity, Bachelor, Digitale Folklore, Labyrinth, Nintendo, Sport, Eltern, Antialiasing, KB talking Pure, Wikipedia, js1k, Deutschland, Django, WiFi, 星のカービィ 夢の泉の物語, Philosophie, Byword, It’s always sunny in Philadelphia, Ikea, Englisch, Kommentare, Existenz, Rechtschreibfehler, iMessage, Ma, Gammeln, Fehlschlag, Bier, Umzug, Tod , Motivation, Robert Bringhurst, Kaleidoscope, Edifier R1900TII, MUJI, PNG Kaffee, Insert Credit Podcast, Textmate, 12.03.2013, The Clancy Brothers, Apps, Compiler, ~1.6180, Gezwitscher, Pizzadienste, funny_cat.gif, Theme Time Radio Hour, Rot, Stuff, Papierkorb, Pilze, Motte, Licht, Englisch,

Das Wesen des Teils

Die Ontologie ist die philosophische Auseinandersetzung mit den Grundstrukturen der Wirklichkeit. Die in diesem Text behandelte Objekt Orientierte Ontologie, kurz *OOO*, setzt Objekte in das Zentrum dieser Betrachtung. In ihr existieren alle Dinge gleich. Autos, Bäume, Atome und Super Mario sind von selber Bedeutung wie der Mensch. Die *OOO* und die mit ihr verbundene *Winzige Ontologie* wird in den Kontext zeitgenössischer Überlegungen gesetzt und mit ihnen verglichen. Des weiteren wird auf die philosophische Praxis an sich eingegangen und das Verfassen von Schriftgut als einzige valide Form der akademischen Auseinandersetzung mit theoretischen Themen in Frage gestellt. Als Beispiel einer solchen alternativen Praxis wird eine Programmiersprache nach den Prinzipien der *OOO* entwickelt und eine Aussicht auf die Visualisierung des Prozesses der Verarbeitung dieser durch den Computer gestellt.

DIE HERKUNFT DER UNIT

Objekte stehen konventionell im Zusammenhang mit dem Subjekt und in der Vergangenheit war es das Zentrum philosophischer Überlegungen. Auch als *erkennendes Ich* bezeichnet, stand es für den Menschen. Quentin Meillassoux fasst diese Sichtweise mit dem Begriff *Korrelationismus* zusammen.[1] In ihr existiert das Sein nur im Zusammenhang zwischen Verstand und Welt. Eine Folge daraus ist, dass, wenn Dinge existieren, sie dies nur für uns Menschen tun, da nur der Mensch über Verstand verfügt. Außerhalb dieses Zusammenhanges kann nichts existieren. *Korrelationismus* wird auch *Post-Kantische Philosophie* genannt, was deutlich macht, dass es sich hierbei um die Sichtweisen handelt, die auf dem Gedankengut von Immanuel Kant aufbauen.

Verfechter dieser Sichtweisen können folgenden Satz nicht ohne weiteres akzeptieren: »Ereignis X fand Y Jahre vor dem Aufkommen der Menschen statt«[2]. Archäologen werden auf diese Weise zu Schöpfern von Sein, indem sie neu gefundene Objekte in unseren Verstand bringen.

Als Gegenbewegung zu diesem *Korrelationismusus* steht der spekulative Realismus, welcher in sich keine philosophische Bewegung ist, sondern der Titel einer Konferenz im Jahre 2007 am Goldsmiths College, University of London war.[3] Dort kamen Philosophen wie Ray Brassier der American University of Beirut, Iain Hamilton Grant der University of the West of England, Graham Harman der American University in Cairo, und Quentin Meillassoux der École normale supérieure in Paris[4] zusammen und behandelten dieses Thema. Der dort anwesende Graham Harman beschreibt in seinem Buch *Tool-Being* die Objekt Orientierte Philosophie, die die Grundlage für die *Unit* von Ian Bogost bildet. Harman basiert seine Philosophie auf das Gedankengut von Martin Heidegger, vor allem auf sein Werk *Sein und Zeit*.

> *Heidegger schlägt vor, dass Dinge als solche unmöglich zu verstehen sind. Stattdessen sind sie mit Zweck verbunden, ein Umstand der das Betiteln von Mundharmonikas oder Tacos als Dinge problematisch macht; Zeug wird zuhanden wenn es in Kontext gestellt wird und vorhanden wenn es von diesem Kontext bricht.*[5]

Harman argumentiert, dass dieses *zuhanden* und *vorhanden* sein für alle Objekte gilt, nicht nur für das *Dasein*, also den Menschen. Objekte existieren daher nicht nur durch die Benutzung des Menschen, sondern durch jede Benutzung, also auch von Objekten durch Objekte. Dadurch wird der Mensch aus dem Zentrum des philosophischen Denkens genommen und mit den Objekten gleichgestellt. Wenn sich zwei Objekte zusammen fügen, dann entsteht ein neues, eigenständiges Objekt, welches ontologisch den selben Wert hat. Zur Verdeutlichung: Der Mensch und die Pistole. Obwohl die Pistole vom Menschen geschaffen wurde, wird sie diesem nicht untergeordnet. Die Hierarchie ist

somit flach, alle Dinge bestehen auf der selben Ebene. Auf diese flache Ontologie wird später im Text nochmals eingegangen. Wenn nun der Mensch eine Pistole aufnimmt, entsteht ein neues Objekt *Mensch mit Pistole*. Die Objekte Mensch und Pistole verlieren dabei nicht ihr Sein, sie bestehen also weiterhin auch als einzelnes Objekt, gleichzeitig sind sie ein neues Objekt. In der Chemie gibt es den Begriff der Emulsion, der für dieses Zusammenführen von Objekten passt. Er ist wie folgt definiert:

> *[Ein] Gemisch zweier normalerweise nicht mischbarer Flüssigkeiten ohne sichtbare Entmischung.*[6]

Die Objekte werden zusammen gemischt, verlieren aber nicht ihre eigentliche Form und erscheinen als ein eigenständiges Objekt.

Damit steht die *Objekt Orientierte Ontologie* genau zwischen dem *Wissenschaftlichen Naturalismus* und dem *Sozialen Relativismus*. Der *Wissenschaftliche Naturalismus* beschreibt, dass Dinge aus immer kleineren Stücken und letztendlich aus bestimmten kleinsten Teilchen bestehen, aus denen alle anderen Dinge erklärt werden können. Der *Sozialen Relativismus* sieht Dinge als Konstrukt menschlichen oder gesellschaftlichen Verhaltens. Laut Bogost befinden sich diese beiden Konzepte in einem Konflikt.[7] Die Verfechter des *Wissenschaftlichen Naturalismus* (im folgenden *Realist*) sehen das wahre Wissen unabhängig von Geschichte und Kontext, während die Anhänger des *Sozialen Relativismus* (im folgenden *Relationist*) genau auf diese Abhängigkeit bestehen. Die *Realisten* glauben an eine Realität außerhalb des menschlichen Verstandes, die nur dafür da ist entdeckt und erforscht zu werden. *Relationisten* glauben die Welt sei ein Konstrukt menschlicher Kultur. Bogost nennt solche Ansätze an die Ontologie *System Operationen*.[8] Er definiert diesen Begriff als »summierende Strukturen die versuchen ein Phänomen, Verhalten oder Zustand in ihrer Gesamtheit zu beschreiben«.[9]

Um Dinge von dem Menschen unabhängig zu betrachten, definiert Levi Bryant die *Flache Ontologie*; Alle Objekte weisen den selben Status auf, keines steht über dem Anderen. In ihr existieren alle Dinge gleich, sind aber nicht gleich. Objekte sind in seiner Theorie breit gefächert; Es sind materielle, sowie immaterielle Entitäten, körperliche Objekte, virtuelle Erscheinungen, fiktionale Dinge oder Abstraktionen. Bücher, Vögel, Bilbo Beutlin, Apple Stores, Konflikte, Atome und typisierte Variablen sind gleich real. Bryant sagt, dass »die Welt nicht existiert«[10] und damit meint er, dass es kein all umfassendes Objekt gibt, welchem alle Objekte innewohnen; Es gibt keine Hierarchie des Seins. Die Welt ist nur ein weiteres Objekt unter den vielen anderen Objekten. Bryant meint, dass diese *Flache Ontologie* den *Wissenschaftlichen Naturalismus* und den *Sozialen Relativismus* zusammen bringen kann, indem sie den Menschen und das Nicht-Menschliche vereint.[11] Er nennt sie aufgrund der Gleichstellung aller Objekte auch die *Demokratie der Objekte*. Man kann sich die *Flache Ontologie* als zweidimensionale Ebene vorstellen, die keinen Ursprung hat und in der alle Objekte gleich verteilt sind. Das bedeutet auch, dass sich Dinge nicht auf andere Dinge reduzieren lassen. Ein Kugelschreiber besteht zwar aus einer Hülle, einem Druckknopf, einer Miene und einer Sprungfeder, aber diese Bestandteile sind nicht das Objekt Kugelschreiber, sondern jeweils ein Objekt für sich. Der *Realist* mag nun die Liste erweitern, somit immer tiefer in die Materialität des Kugelschreibers blicken, und zu einem für ihn zufriedenstellendem Ergebnis kommen. Der *Relationist* schaut sich vielleicht das Verhältnis des Menschen zum Kugelschreiber und seine Bedeutung in der Geschichte der Menschheit an. Beide Sichtweisen sind in der *Flachen Ontologie* nicht ausreichend.

Bogost baut auf der *Flachen Ontologie* auf und entwickelt die *Winzige Ontologie*.[12] Dabei bezieht er Bruno Latours *actor-network* Theorie mit ein. Das Beispiel des *Mensch mit Pistole* Objektes wurde aus den Ausführungen der *actor-network* Theorie übernommen und ist gleichermaßen für beide Theorien gültig. Diese sieht alle Dinge in einem Netzwerk aus Beziehungen zu-

einander. Allerdings ist die Interaktion zwischen den Dingen nicht in ihrem Wesen verankert, sondern außerhalb ihrer. *Netzwerk* als Begriff deutet ein sehr strukturiertes Verhältnis der Objekte zueinander an. Latour hat in seinen späteren Beschreibungen den Begriff des Netzwerkes durch *imbroglio*, zu deutsch *Verwicklung*, ausgetauscht. Damit wird das Geordnete aus den Verhältnissen genommen und in eine gewisse Unordnung gebracht. John Law nutzt den Begriff *Mess*, zu deutsch *Durcheinander*, anstelle des *Netzwerkes*, um das Problem des Geordneten zu umgehen.[13] Beide Begriffe versuchen das Problem der Struktur zu umgehen. Egal ob es sich um die Ordnung oder eben die Unordnung handelt, es bleibt immer eine Struktur bestehen.

Die *Winzige Ontologie* besitzt eine Dimension weniger als die *Flache Ontologie*. Alle Dinge existieren in einem eindimensionalen Punkt. Aus diesem Grund wird hier der Begriff *Winzig* benutzt. Es geht um die Vereinfachung des Seins und dessen kompakteste und schmuckloseste Darstellung. Harman vergleicht das Ding an sich mit einem Schwarzen Loch: »Jedes Objekt ist nicht nur durch ein nichtssagendes Schild von den Dingen außerhalb geschützt, sondern beherbergt und pflegt ein ausbrechendes höllisches Universum in sich«[14]. Schwarze Löcher sind in ihrer ursprünglichen Definition astronomische Objekte, deren Dichte so groß ist, dass die dadurch entstehende Gravitation nichts nach außen dringen lässt und sogar die Zeit moduliert.[15] Nikodem Poplawski stellte die Theorie auf, dass die Dichte im Innern des Schwarzen Loches dafür sorgt, dass die Gravitation umgedreht wird und sich das Schwarze Loch aus diesem heraus ausdehnt.[16] Daraus schließt er, dass auch unser Universum sich in einem solchen Schwarzen Loch befinden könnte.

Die Gleichstellung aller Dinge hat eine weitere Konsequenz: Das Wesen eines Objektes ist ein Objekt. So steht das Wesen eines Objektes neben – *Flache Ontologie* – oder im selben Punkt – *Winzige Ontologie* – wie das Dasein des selben Objektes.

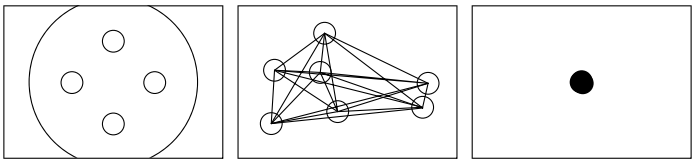


Abbildung 1. Von links nach rechts: Wissenschaftlicher Naturalismus, Sozialer Relativismus, Winzige Ontologie.

Das *Objekt Orientierte* aus der *Objekt Orientierten Ontologie* ist der Informatik entnommen. Es beschreibt ein bestimmtes Programmierparadigma das 1993 von Alan Kay wie folgt definiert wurde:

> *1. Alles ist ein Objekt,*
> *2. Objekte kommunizieren durch das Senden und Empfangen von Nachrichten (welche aus Objekten bestehen),*
> *3. Objekte haben ihren eigenen Speicher (strukturiert als Objekte),*
> *4. Jedes Objekt ist die Instanz einer Klasse (welche ein Objekt sein muss),*
> *5. Die Klasse beinhaltet das Verhalten aller ihrer Instanzen (in der Form von Objekten in einer Programmliste),*
> *6. Um eine Programmliste auszuführen, wird die Ausführungkontrolle dem ersten Objekt gegeben und das verbleibende als dessen Nachricht behandelt.*[17]

Im ersten Punkt stimmen beide Definitionen überein. Ab Punkt vier wird die Definition dann sehr spezifisch für die Informatik und weicht von der Definition der *Winzigen Ontologie* ab. Das Verhältnis zwischen Klassen und Instanzen deutet eine Hierarchie an und macht die Instanz von der Klasse abhängig, was den eigenständigen Objekten der *OOO* widerspricht. Punkt zwei und drei sind noch weiter für die *Winzige Ontologie* interessant. Punkt zwei spricht etwas an, das in dieser Abhandlung der *OOO* noch nicht beachtet wurde: Das Verhalten und die Interaktion der Objekte zueinander. Bogost verwendet als Beschreibung

hierfür den Begriff *Operation*.[18] *Operation* wird in der System Theorie als »grundlegender Prozess« beschrieben, »der eine oder mehrere Eingaben annimmt und auf diese eine Transformation anwendet«[19]. Die Objekte in der *Winzigen Ontologie* operieren ständig. Sie integrieren sich mit sich selbst und verwickeln sich mit anderen, agieren und reagieren auf Eigenschaften und Zustände während sie etwas verschweigen.[20] Die Operation ist immer fraktal[21] – gebrochen und sich selbst ähnlich. »Diese Dinge machen sich Gedanken über einander ohne jemals Bestätigung zu bekommen«[22]. Diese nicht vorhandene Bestätigung wird auch in Punkt drei der Definition der *Objekt Orientierte Programmierung* angesprochen. Objekte haben ihren eigenen Speicher, auf den nur sie Selbst zugreifen können. Der Austausch zwischen Objekten findet daher nicht direkt statt, sondern immer über die oben genannten Operationen.

Der Begriff Objekt hat mehrere Bedeutungen, die ihn für die *Winzige Ontologie* zwar nicht unpassend machen, er trotzdem aber irreführend sein kann. Bei dem Objekt schwingt ein Subjekt mit, welches an sich kein Problem darstellt, allerdings wird es meist genutzt, um gerade beides von einander zu trennen. In einer Ontologie, in der alles ein Objekt ist, kann nichts existieren, das ein *Nicht-Objekt* ist. Zumal das Subjekt häufig dem Mensch zugeordnet wird und in der "Winzigen Ontologie" der Mensch auch ein Objekt ist. Weitaus gravierender ist, dass ein Objekt Materialität impliziert. So wird es jedenfalls definiert:

> *Gegenstand, auf den sich jemand bezieht, auf den das Denken oder Handeln ausgerichtet ist.*[23]

All dies sind Gründe, die Bogost dazu bewegt haben, ein neues Wort für Objekt zu finden, wie es im Kontext der *Winzigen Ontologie* gebraucht wird. Sein Resultat ist die *Unit*.

> *[…] Zum einen ist eine Unit isoliert und einzigartig. Zum anderen umfasst eine Unit ein System. Zum weiteren ist eine Unit teil eines anderen Systems – oft vielen anderen Systemen.*[24]

Daraus ergibt sich eine weitere Eigenschaft des Wesens der *Unit*; Sie ist nicht nur sie selbst, sondern auch etwas anderes und baut somit ein Paradoxon auf. Sie ist nicht starr, sondern in ständiger Bewegung; Sie operiert. Das Operieren mit anderen *Units* gibt diesen einen Teil ihres Wesens preis, aber nie das Ganze. So ist es nicht möglich, den inneren Wesenskern einer *Unit* in seiner Vollständigkeit zu erfassen; Uns bleibt nur das, was sie uns mitteilt. Betrachtet man nun die Definition des Wesens von Heidegger, verschiebt sich der Begriff des Wesens von dem Unbekannten des Inneren der *Unit* zu der *Operation* der *Unit*. »Die Technik ist nicht das Gleiche wie das Wesen der Technik «[25]. Das Wesen ist also nicht die Sache an sich, sondern etwas anderes. Im Falle Heideggers ist es das, was wesentlich zur Bestimmung einer Sache ist. Einen Hammer macht nicht aus, dass er aus einem Stück Holz und einem Stück Eisen besteht, sondern die Tatsache, dass man mit ihm hämmern kann. Genauso liegt im Wesen der *Unit* nicht die *Unit* an sich, sondern die *Unit Operation*.

ZURÜCK ZUM TEIL

Die direkte Übersetzung des Begriffes *Unit* in die deutsche Sprache wäre *Einheit* gewesen. Die Einheit ist ein mächtiges Wort, es verinnerlicht etwas, es besteht aus kleineren Stücken, es ist ein Ganzes, es ist die göttliche Einheit, sie ist vollkommen. All dies ist Ballast, der dem Eigentlichen nicht zugute kommt. Ein Teil ist kleiner, unbedeutender, ein Ding unter Vielen, eigenständig, aber auch Teil etwas Anderem. Es baut dasselbe Paradoxon wie die Objekte der *Winzigen Ontologie* auf und ist damit ein ebenbürtiger, wenn nicht sogar passenderer Begriff als *Unit*.

Aus der *Unit Operation* wird somit die Teiloperation. Bei dieser einfachen Übersetzung kommt ein weiteres Merkmal der Operationen zum Vorschein. Teile operieren nicht im Ganzen, auch ihre Operationen sind in Teile zerlegt. Sie geben sich nie ganz zu erkennen, behalten etwas und bestimmen was operiert. Wie eine

solche Operation aussieht, hängt von den operierenden Teilen ab. In dem Buch *A New Kind of Science* stellt Stephen Wolfram die These auf, dass die Welt von Natur aus digital ist und somit deren innewohnende Prozesse berechenbar sind. Er argumentiert, dass diese Prozesse experimentell erforscht werden müssen und die Resultate Rückschlüsse auf die Natur zulassen. Wolframs Auffassung der Dinge weicht von der *Winzigen Ontologie* ab, aber seine Überlegungen in Richtung der Prozesse lassen einen Einblick in die Operation der Teile werfen. Im Fokus von Wolframs Experimenten steht das *Einfache Programm*, er definiert dieses wie folgt:

> *1. Seine Operation kann komplett durch eine einfache grafische Illustration beschrieben werden.*
> *2. Es kann komplett mit ein paar Sätzen menschlicher Sprache beschrieben werden.*
> *3. Es kann durch wenige Zeilen Code in einer Programmiersprache implementiert werden.*
> *4. Die Anzahl der Variationen ist klein genug, dass alle berechnet werden können.*[26]

Einfache zelluläre Automaten sind Teil dieser *Einfachen Programme*. Ihr nächster Zustand geht aus dem aktuellen Zustand der Zelle und den Zustand der direkten Nachbarzellen hervor. Daraus ergeben sich acht verschiedene Konstellationen, die zu 256 Regeln führen, von denen sich 88 in ihrem Verhalten unterscheiden. Die Regel 110 ist von besonderer Bedeutung, da sich diese als turing-vollständig erwiesen hat.

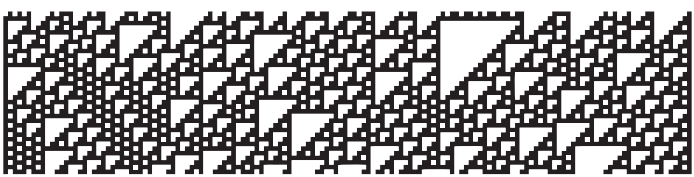


Abbildung 2. Regel 110 des Zellulären Automaten mit einem zufälligen Anfangszustand, einer Breite von 610 Zellen und 35 Zuständen.

Turing-vollständig bedeutet für ein System, dass es in der Lage ist, sämtliche Funktionen zu berechnen, die eine universelle Turing Maschine berechnen kann und ist damit gleichbedeutend mit seiner universellen Programmierbarkeit.[27] Je nach Ausgangsform, ist die Regel 110 also in der Lage die Funktionsweise von modernen Computern zu emulieren.

Das Verhalten von *Einfachen Programmen* besteht aus wenigen Teilen. Das Resultat dieses Verhaltens ist aber so komplex, dass man nicht mehr von der Summe der einzelnen Teile sprechen kann, sondern etwas völlig neues entsteht. Dieses Phänomen der *Einfachen Programme* kann man als Analogie zum Verhalten der Teiloperationen sehen. Wenn aus einer Operation mehr als nur die Summe ihrer enthaltenen Teile entsteht, spricht man auch von Emergenz.

Wenn wir nun nach dem Wesen des Teils fragen, geraten wir in eine Zwickmühle. Einerseits können wir sagen, dass das Wesen des Teils in seinen Operationen liegt, anderseits ist diese Antwort nicht ganz zufriedenstellend. Wir bekommen eine Beschreibung dessen, aber sagt diese alles über das Wesen aus? Da die Teiloperationen, wie eben festgestellt wurde, ein emergentes System sind, besteht das Wesen nicht nur aus den beschreibbaren Einzelteilen, sondern aus mehr. Wie bekommt man also Einblick in das Wesen eines Teils?

Carl Sagan erklärt in der Folge *Am Rande der Ewigkeit* seiner Fernsehserie *Unser Kosmos*[28] die vierte Dimension. Von der zweiten Dimension aus gesehen, erklärt er mit Hilfe der Bewohner von Flatland[29], wie diese ein Wesen aus der dritten Dimension wahrnehmen. Wir als Menschen – Wesen der dritten Dimension – sind in der Lage in der zweiten Dimension zu denken. Man nehme ein Blatt Papier, schneide daraus verschiedene Formen, denkt sich die Höhe des Papiers weg und schon hat man eine recht präzise Darstellung von Objekten in der zweiten Dimension. Wenn nun ein Objekt der zweiten Dimension ein Objekt der dritten Dimension betrachtet, kann es dieses nicht in seinem Ganzen erfassen, da es nur in zwei Dimensionen sehen kann. Es

sieht nur einen Querschnitt des anderen Objektes. Genauso verhält es sich mit uns Menschen und dem Erkennen der vierten Dimension. Sagan hält ein dreidimensionales Modell eines vierdimensionalen Hyperwürfels so ins Licht, dass sich auf seinem Tisch eine exakte Repräsentation des Hyperwürfels in zwei Dimensionen bildet. Er benutzt die Analogie des Schattens um die vierte Dimension für uns wahrnehmbar und denkbar zu machen. Für Bogost ist dies der einzige Weg, um Objekte beschreiben zu können.[30] Eine Analogie ist kein objektives Werkzeug, das ist für Bogost kein Problem, sondern genau so gewollt.

WERKEN

Rain drops on roses and whiskers on kittens,
Bright copper kettles and warm woolen mittens,
Brown paper packages tied up with strings…
These are a few of my favorite things.

Cream-colored ponies and crisp apple strudles,
Door bells and sleigh bells and schnitzel with noodles,
Wild geese that fly with the moon on their wings…
These are a few of my favorite things.

Girls in white dresses with blue satin sashes,
Snowflakes that stay on my nose and eyelashes,
Silver white winters that melt into spring-
These are a few of my favorite things.

When the dog bites
When the bee stings
When I'm feeling bad…
I simply remember my favorite things
And then I don't feel so bad

— Richard Rodgers: My Favorite Things[31]

In Listen werden Dinge durch ein einfaches Komma getrennt. Die einzelnen Dinge müssen nicht einmal aus Beschreibungen, wie im hier zitierten Fall von *My Favorite Things* bestehen, es kann einfach eine Aneinanderreihung von Begriffen sein, die in ihrem Ganzen einem bestimmten Zweck dienen. In dieser Form der einzelnen Begriffe bilden sie die grundlegendste Form der Ontografie. Ontografie wird von Bogost als »eine ästhetische Set Theorie, in der eine bestimmte Kombination aufgrund ihrer bloßen Existenz gefeiert wird«[32] beschrieben. Laut Michael Lynch ist es »die beschreibende Alternative zu ihrem philosophischen Gegenstück (Ontologie)«[33]. Am Anfang dieses Buches befindet sich eine solche Liste. Sie ist ein Beispiel für den Inhalt des Buches, ohne dass dieser erklärt wird. Es sind die Objekte, welche eine Rolle für das Buch spielen, und wie in einem Dunstkreis um das Thema schweben. Solche Listen erscheinen häufiger in den Werken von Bruno Latour und so gab Bogost ihnen den Namen *Latour litanies*.[34] »Listen von Objekten ohne Erklärung haben eine besondere Macht, indem sie die philosophische Aufgabe erledigen, unsere Aufmerksamkeit auf diese zu lenken.«[35]

Die Liste ist nicht die einzige Methode der Ontografie, aber das Prinzip der Zusammenstellung von losen Objekten bleibt bestehen. Es gibt auch visuelle Ontografien, zum Beispiel bestehend aus Illustrationen. In dem Buch *Siteless* von François Blanciak befinden sich über 1000 Zeichnungen von Objekten verschiedenster Form und Größe, die ebenbürtig in den selben Ausmaßen dargestellt werden.

> *Diese Studie entwickelt als kreative Alternative zur kritisch akademischen Literatur eine vorausblickende Serie an Formen, die sich auf den Kern der Architektur, das Gebäude als Einheit, und die Klarheit des Ausdruckes seiner generativen Idee konzentrieren. Als Resultat wurde in den kommenden Kapiteln Text durch Form ersetzt.*[36]

Die Ontografie bietet einen alternativen Weg, Philosophie zu praktizieren. Eine Alternative zu dem traditionellen Weg des

Verfassens von Schriften. Aber sie ist nicht die einzige Alternative. Unter dem Begriff *Carpentry* zu deutsch *Werken* verfasst Bogost ein Prinzip, welches dem Philosophen erlaubt, seine *Hände schmutzig* zu machen und Artefakte zu konstruieren. »Der Werkende muss sich mit dem Widerstand des Materials seiner Wahl auseinandersetzen, was das Objekt selber zur Philosophie werden lässt«[37]. Das bedeutet nicht, dass jedes Produkt menschlichen Werkens Teil des philosophischen Diskurses werden soll, sondern nur, dass die Dinge, die in Hinsicht auf dessen erstellt wurden, einen validen Part im Ganzen spielen. »[…] nicht wie Werkzeuge und Kunst ist philosophisches Werken unter Beachtung der Philosophie zu handhaben: es mag einer Vielzahl an anderen produktiven oder ästhetischen Zwecken dienen, […], aber es ist erst als Theorie konstruiert, oder als Experiment oder als Frage – etwas das operiert werden kann. Werken ist philosophisches Labor Zubehör«[38]. Durch das *Werken* versucht man die Teiloperationen eines anderen Objektes zu kopieren. Dadurch erlangt man Einblick in die Erfahrung von Objekten, die man zwar nicht verstehen kann, aber der Einblick kann ausreichen, um ein zufrieden stellendes Bild derer aufzuzeigen.

Praxis

Mein Handwerk ist das Programmieren und daher macht es nur Sinn, sich damit die *Hände schmutzig* zu machen und die Tiefen der Philosophie zu erforschen. Das Programmieren an sich hat die interessante Eigenschaft, dass man Systeme modelliert und diese sofort vom Computer simulieren lassen kann. Dazu muss man sie erst abstrahieren – auf ihre berechenbaren Komponenten herunter brechen – und konkretisieren – diesen Komponenten Werte zuweisen – damit der Computer sie versteht. Der produzierte Code ist in seiner Form ambivalent, da er von zwei Entitäten gelesen und verstanden werden muss; Dem Menschen und dem Computer. Des weiteren ist das Programmieren nur *Mittel zum Zweck*. Es geht nicht darum Code zu schreiben, sondern den Computer dazu zu bringen, bestimmte Dinge zu tun.

Ich habe dies mittels der Programmiersprache Javascript in den interaktiven Versionen von Abbildung 1 durch exerziert.[39] Der hier dargestellte Code besteht aus Pseudocode, einer fiktiven Programmiersprache, die ermöglicht, dass sich auf die Logik konzentriert werden kann. Der Code zu jeder der drei Abbildungen ist nach ihrem darstellendem Prinzip modelliert.

WISSENSCHAFTLICHER NATURALISMUS

> *Dinge bestehen aus kleinsten Teilchen und die Welt ist dafür da, durch den Menschen entdeckt zu werden.*

Die Welt der *Realisten* lässt sich also wie folgt beschreiben:

```
definiere Welt als WissenschaftlicherNaturalismus {
  elementZumInspizieren = neues Element()

  tiefsterZoom = 10
  aktuellerZoom = 1
}
```

Die vereinfachte Welt des *Wissenschaftlichen Naturalismus* besteht aus einem Element und zwei Angaben zum aktuellen und maximalen Detailgrad. Klickt man nun auf ein Element, wird der Detailgrad erhöht und die im angeklickten Element enthaltenen Elemente angezeigt. Dies lässt sich so oft wiederholen, bis man den größten Detailgrad erreicht hat und man die Elemente betrachtet, aus denen alle anderen Elemente bestehen.

Wie schaut nun ein solches Element aus?

```
definiere Element() {
  farbe = zufällig aus [Farbbereich]
  bestandteile = neue Elemente()
}
```

Es besteht aus einer zufälligen Farbe und einer beliebigen Anzahl weiterer Elemente. Die Farbe dient als Identifikation für das Element, um es von den anderen zu unterscheiden.

SOZIALER RELATIVISMUS

> *Dinge sind Konstrukt menschlicher Kultur und das Verhalten derer wird durch die Gesellschaft bestimmt.*

```
definiere Welt als SozialerRelativismus {
    Gesellschaft = neue Aktoren( zufällig aus [Attraktor,
                                                Repulsor] )
  aktiverAktor = zufällig aus Gesellschaft
}
```

Die Welt des *Soziale Relativismus* besteht aus einer Gesellschaft mit beliebig vielen Aktoren, die als Verhalten entweder ein *Attraktor* oder ein *Repulsor* sind. Die Gesellschaft bestimmt zufällig welches der beiden Verhaltensweisen benutzt wird. *Attraktoren* sind Aktoren, die andere Aktoren in der Gesellschaft anziehen. Die *Repulsoren* sind genau das Gegenteil der Aktoren, sie stoßen die anderen Aktoren ab. Zusätzlich wird noch zufällig ein Aktor ausgewählt, der von der Person, die mit dem System interagiert, per Maus gesteuert wird.

Der Aktor in seiner vereinfachten Darstellung:

```
definiere Aktor( Verhalten ) {
    verhalten = neues Verhalten()
    farbe = farbe von [verhalten]
}
```

Um die beiden verschiedenen Typen von Verhaltensweisen besser zu unterscheiden, wird dem Aktor die, dem Verhalten zugewiesene Farbe, zugeordnet.

Anmerkung zur Darstellung: Dass sich die einzelnen Aktoren in vertikale Formationen anordnen ist ein Produkt aus der Kollision und den Kräften, die auf die einzelnen Aktoren ausgeübt werden. Es ist das Produkt dieses Systems und auch nur Ausdruck genau dieser Konstellation. Es soll nicht dafür sprechen, dass Systeme in dem *Sozialen Relativismus* stabile, geordnete Konstellationen suchen.

```
definiere WinzigeOntologie {
  units = neue Units()
}
```

Die Welt der *Winzigen Ontologie* ist im Vergleich zu den anderen beiden sehr einfach. Sie besteht lediglich aus den enthaltenen *Units*.

```
definiere Unit() {
  geheimnis = neues Geheimnis()

  definiere operation( objekt ) {
    position = position + position von objekt
  }
}
```

Die *Unit* besteht aus zwei Teilen, zum einen aus dem nur ihr bekannten Geheimnis, das ihr innewohnt, und einer Funktion, die die Operation mit anderen *Units* beschreibt. In diesem Fall wird die Position der *Unit* beeinflusst.

Alle drei Beispiele sind als Analogie zu den Systemen, die sie darstellen, auf zwei Ebenen zu verstehen; Der Ebene der Darstellung und der Ebene der Programmierung.

part

Der Pseudocode vom vorherigen Kapitel ist nicht ausführbar und auch die Javascript Version dessen muss Einschränkungen eingehen. Wenn man nach einer genaueren Repräsentation eines philosophischen Systems in Maschinen interpretierbarem Code verlangt, wird man unter den vielen schon vorhandenen Programmiersprachen keine finden, die genau passend ist, da diese nach anderen Paradigmen gestaltet sind. Wenn nach etwas passendem gefordert wird und es noch nichts dergleichen gibt, muss man was passendes erschaffen. Wie könnte also eine Programmiersprache aussehen, die nach den Prinzipien der *OOO* gestaltet ist?

Bevor dies Frage beantwortet wird, müssen ein paar Rahmenbedingungen festgelegt werden. Die Sprache soll keinem anderen Zweck dienen, als die Prinzipien der *OOO* darzustellen. Das heißt, dass einige Dinge bei der Implementierung missachtet werden können und keine Optimierungen auf technischer Ebene stattfinden müssen. Es geht nicht darum neue Wege der Darstellung für Algorithmen zu finden oder dem Programmierer mehr Freiheiten zu bieten. Sie muss aber, um eine Programmiersprache zu sein, vom Computer interpretiert und ausgeführt werden können. Darüber hinaus sollte die Sprache möglichst kompakt in ihrem Umfang sein. Das erlaubt eine einfachere Implementierung und zwingt die vorhandenen Elemente dazu, möglichst flexibel zu sein. So könnte man im Sinne der *Einfachen Programme* von einer *Einfachen Programmiersprache* sprechen.

Das Konstruieren von etwas Neuem hat neben der völligen Anpassung an ein Thema noch weitere Vorteile. Es entbirgt etwas vom Wesen des konstruierten Objektes. Der Möglichkeitenraum erweitert sich gegenüber dem eines vorgefertigten Objektes. Im

Falle von Programmiersprachen entfernt es eine Abstraktionsschicht und gewährt Einblick in das Wesen derer.

Die Syntax ist wie im vorherigen Kapitel bemerkt für zwei Objekte von Bedeutung, dem Menschen und den Computer. Beide müssen den ihnen vorliegenden Code verstehen. Da Programmiersprachen von Menschen geschaffen werden, liegt der Fokus häufig auf der Lesbarkeit für den Menschen. Im Falle der *OOO* sind die beiden Objekte gleichwertig, die Sprache sollte daher keines der Beiden favorisieren. Ein Beispiel für eine maschinennahe Sprache wäre die Assemblersprache für die x86 Architektur. Die Addition von vier Werten, [a, b, c, d] sieht in diese wie folgt aus:

```
mov eax, [a]
add eax, [b]
add eax, [c]
add eax, [d]
```

Eine für den Menschen besser leserliche Programmiersprache ist *Ruby*:

```
summe = a + b + c + d
```

Programmieren findet bis auf wenige Ausnahmen in Englisch statt und daher vergebe ich, obwohl der Text dieses Buches in Deutsch verfasst ist, der Programmiersprache den Namen *Part*, englisch für *Teil*.

SYNTAX

Die Syntax von *Part* ist folgendermaßen definiert:

```
(Operation Körper)
```

Alles, was in Klammern gesetzt ist, stellt ein Teil dar. Ein Teil benötigt keinen Inhalt, so dass es sich bei `()` um ein

valides Teil handelt. Wenn eine Operation vorhanden ist, kann der Körper nicht leer sein, ein Körper kann aber ohne Operation existieren. Die Bedingungen für die jeweiligen Operationen werden im nachfolgenden Kapitel erklärt. Ein Teil gibt immer, sofern es nicht operiert, den letzten validen Ausdruck aus dem Körper zurück. `(1 2 3 4)` gibt demnach `4` zurück. Wenn das Teil nur aus einem Ausdruck besteht, kann zu Gunsten der Lesbarkeit auf die Klammern verzichtet werden.

Alles, was in einer Zeile nach einem `#` folgt, wird als Kommentar für den Menschen angesehen und vom Computer ignoriert.

OPERATIONEN

Es gibt neben den arithmetischen Operationen, den bitweisen Operationen und den Vergleichsoperationen nur fünf weitere: `def`, `cond`, `while`, `num` und `array`.

`def` definiert ein Teil und gibt ihm einen Namen. Der Syntax dafür sieht wie folgt aus:

```
(def Name(Parameter) (Körper))
```

Die Parameter werden dafür verwendet mit dem Teil zu operieren. Dies geschieht indem man beim Aufruf ein Teil mit der selben Anzahl an Parametern übergibt. Jede Definition beinhaltet einen Parameter-Teil. Teile operieren also mit anderen Teilen über Teile. Die Parameter-Teile sind nur im jeweiligen Körper ihrer Definition gültig und müssen mit dem entsprechenden Datentypen ausgezeichnet werden.

```
(def addOne(num a)
  (+ a 1))
(addOne(3))
# Ausgabe: 4
```

Als Namen sind Zeichenfolgen aus alphanumerischen Zeichen valide, sofern sie mit einem Buchstaben anfangen. Groß- und Kleinschreibung wird nicht beachtet. **def**, **cond**, **while**, **num** und **array** sind reservierte Wörter und dürfen nicht als Namen verwendet werden.

Die arithmetischen Operationen führen ihre jeweilige Operation auf die folgenden numerischen Teile aus. Sie bestehen aus Addition (+), Subtraktion (-), Multiplikation (*) und Division (/).

```
(+ 1 2 3)
# Ausgabe: 6
```

Das Verschachteln verschiedener arithmetischen Operationen ist möglich:

```
(- 3 (* 2 3))
# Ausgabe: -3
```

Bitweise Operationen können nur auf natürliche Zahlen angewendet werden. Sie führen Operationen auf die binäre Repräsentation einer Zahl aus. Unterstützt werden *UND* (&), *ODER* (|), *Exklusives ODER* (^), *NICHT* (~) und *Shift nach Links* (<<) sowie *Shift nach rechts* (>>).

```
(| 4 3)
# 4: 0100
# 9: 1001
# A: 1101 = 13
```

Vergleichsoperatoren vergleichen den Inhalt ihres Körpers miteinander und geben entweder eine 0, wenn der Vergleich nicht wahr ist oder eine 1, wenn der Vergleich wahr ist, zurück. Sie bestehen aus *Ist-Gleich* (==), *Ist-Nicht-Gleich* (!=), *Ist-Größer* (>), *Ist-Größer-Gleich* (>=), *Ist-Kleiner* (<), *Ist-Kleiner-Gleich* (<=), dem logischen UND (&&) und dem logischen ODER (||). Es ist nicht erlaubt zwei Teile von unterschiedlichem Datentyp zu

unterscheiden und sie können nicht auf **Arrays** angewandt werden, nur auf den Inhalt derer.

```
(> 5 3)
# Ausgabe: 1

(&& (< 4 9) (== 3 6))
# Ausgabe: 0
```

Zuweisungsoperatoren verändern den Wert eines gegebenen Teils. Dies funktioniert nur mit Teilen, welche mit Namen definiert sind.

```
(num nine 9 (= nine 3))
# Ausgabe: 3
```

Die **cond** Operation ermöglicht es, eine Bedingung auf ihren Wahrheitsgehalt zu überprüfen und darauf zu reagieren.

```
(cond (< 3 4) 5 6)
# Ausgabe: 5

(cond (== 3 4)
  (+ 3 4)
  (- 3 4))
```

Die **while** Operation wiederholt ihren Körper solange, bis ihre Bedingung nicht mehr erfüllt ist.

```
(num index 5
  (while (> index 0)
    (-= index 1)))
# Zählt index von 5 zu 1 runter
```

DATENTYPEN

Alles ist ein Teil, aber damit der Computer etwas damit anfangen kann, müssen diese Teile auf Datentypen herunter gebrochen werden, die der Computer versteht. Numerischen Teile werden mit dem Datentyp `Number` deklariert und Sammlungen von Nummern werden als `Array` hinterlegt.

Numerische Teile können als Natürliche Zahl oder als Gebrochene Zahl beschrieben werden. Die Trennung bei gebrochenen Zahlen erfolgt durch einen Punkt; Bsp.: `1.23`. Die Definition der Zahl darf nicht mit einem Punkt enden. Eine Definition sieht dabei wie folgt aus:

```
(num nine 9 (+ 1 nine))
# Ausgabe 10
```

Wie bei der Definition der Funktion hat die Definition einer Nummer einen Körper und diese Nummer ist nur innerhalb diesen Körpers gültig.

Es können auch mehrere Nummern gleichzeitig definiert werden. Dazu müssen die Definitionen durch ein Komma getrennt werden.

```
(num n 1, f 2.2 (+ n f))
```

`Arrays` haben immer eine feste Größe und werden wie folgt festgelegt: `(array a länge (körper))`. Die Länge muss dabei eine natürliche Zahl sein. Ein Element aus dem Array spricht man wie folgt an: `(array[elementNummer])`.

BEISPIELE

Wie sich das in einem funktionierenden Programm verhält, wird am Beispiel der Berechnung der Fakultät und einem einfachen zellulären Automaten dargelegt.

Die Fakultät ist eine Funktion, die das Produkt aller Zahlen kleiner und gleich einer Zahl berechnet. Es wird häufig als einfaches Beispiel für rekursive Funktionen benutzt.

```
(def factorial (n)
  (cond (<= n 1)
    1
    (* n (factorial(- n 1)))))
```

Es folgt der Code für einen Zellulären Automaten:

```
(def printBlock() print( '▉' ) print( '▉' ))
(def printSpace() print( ' ' ) print( ' ' ))

(def automata(array row, num rule, num width, num height)
  (num y 0, x 0
    (array nextRow 200
      (while (< y height)
        (= x 0)
        (while (< x width)
          (num state 0
            (= state (+ state (cond (== row[x-1] 1) 4 0)))
            (= state (+ state (cond (== row[x  ] 1) 2 0)))
            (= state (+ state (cond (== row[x+1] 1) 1 0)))
            (= nextRow[x] (== (& (>> rule state) 1)) 1)
           (cond (== nextRow[x] 1) printBlock() printSpace())
            (= x (+ x 1))))
          (= row nextRow)
          (println(''))
          (= y (+ y 1))))))

(num width 200
  (array firstRow 200
    (= firstRow[(/ width 2)] 1)
    (automata(firstRow 110 width 100))))
```

Compiler

Nun wurde die Programmiersprache *Part* definiert, sie ist aber noch keine turing-vollständige Sprache, da sie von keinem Computer verstanden werden kann. Computer können zwar Text einlesen, aber sie können ihn nicht von sich aus interpretieren. Dazu benötigen sie einen für die Programmiersprache angepassten Compiler. Ein Compiler ist ein Programm, das Code einliest und in eine maschinell lesbare Form bringt. Grundsätzlich durchgeht der Compiler zwei verschiedene Phasen: Die Analyse und Abstraktion des Codes sowie die Erzeugung des Programms. In beiden Phasen lassen sich Optimierungen durchführen, welche für sich ein eigenes Gebiet sind, aber im Falle dieser Bearbeitung des Themas keine Rolle spielen sollen. Der hier beschriebene Prozess ist nicht für alle Compiler gültig, von der generellen Funktionsweise ist er aber gängigen Compilern nachempfunden und in seiner Komplexität reduziert.

Beide Phasen sind nochmals in zwei Schritte unterteilt. Es ist möglich diese vier Schritte in eigenen Programmen um zu setzten, die geringe Komplexität von *Part* und der Verzicht auf Optimierung ermöglicht es aber alles in einem Programm unter zu bringen.

LEXER

Im ersten Schritt wird eine *lexikalischer Analyse* durchgeführt. Der Name *Lexer* ist eine Abkürzung von *lexikalischer Scanner*. Der dem Programm übergebene Quellcode wird eingelesen und die einzelnen Teile nach der für die Programmiersprache definierten Grammatik in die entsprechenden *Tokens* unterteilt. Ein *Token* basiert aus einem Typ und der dazugehörigen Zeichenfolge.

Die Zeichenfolge `123` hätte im Falle von *Part* den Typ `Number`. In diesem Schritt werden meist überflüssige Zeichenfolgen aus dem Quellcode entfernt. Zu diesen gehören Leerzeichen und Kommentare.

PARSER

Der *Parser* benutzt die vom *Lexer* erstellten *Tokens* um einen *Abstrakten Syntax Baum*, kurz *AST*, zu generieren. Dieser stellt den Zusammenhang der einzelnen *Tokens* zueinander dem Syntax entsprechend dar. Die Definition des Teils `(+ 1 2)` wird zu folgendem Baum:

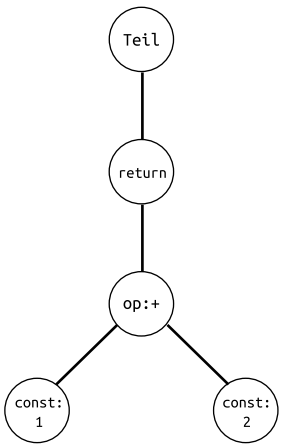


Abbildung 3. Der AST des Teils `(+ 1 2)`.

LLVM

Die letzten beiden Schritte übernimmt im Falle von *Part LLVM*[40]. *LLVM* ist eine Sammlung von Werkzeugen zur Erstellung und Optimierung von ausführbaren Programmen. Im ersten Schritt wird

dabei der *AST* in die *LLVM IR* Assembler Sprache übersetzt, die wiederum dazu genutzt wird um das Programm für die verschiedenen unterstützenden Architekturen lauffähig zu machen. Sofern es keine weiteren Abhängigkeiten der Sprache zu einer bestimmten Architektur bzw. Plattform gibt, kann der selbe Quellcode auf verschiedenen Plattformen ausgeführt werden, ohne das weitere Anpassungen notwenig sind. So lässt sich zum Beispiel ein Quellcode zu einem *x86*[41] und einem *arm v7*[42] kompatiblen Programm kompilieren.

Der Vollständigkeit halber sei hier der aus dem obigen Beispiel-AST generierte *LLVM-IR* Code erwähnt:

```
define double @0() {
entry:
  %addtmp = fadd double 1.000000e+00, 2.000000e+00
  ret double %addtmp
}
```

AUSSICHT

Das Ende der schriftlichen Auseinandersetzung ist erreicht. Aber nicht das Ende des Bachelor Reportes. Dieser schriftlich verfasste Teil dient als Einführung und Grundlage für den, aus Mangel an passenden Worten, als *Nicht-Schriftlichen Teil* benannten Teil des Bachelor Reportes. Dieser wird aus der Implementierung der hier entwickelten Programmiersprache *Part* bestehen, sowie einem Programm, das einen Blick hinter die Kulissen des Compilers ermöglicht. Es wird wie eine Explosionsgrafik die innewohnenden Prozesse des Compilers darlegen.

Quellenangaben

LITERATURVERZEICHNIS

1 Ian Bogost: Alien Phenomenology or what it's like to be a thing. Minneapolis, 2012, S. 4

2 Ian Bogost: Alien Phenomenology: S. 4

3 Ian Bogost: Alien Phenomenology: S. 5

4 http://en.wikipedia.org/wiki/Speculative_Realism, zuletzt gesehen am 12.03.2012

5 Ian Bogost: Alien Phenomenology: S. 5, wurde vom Verfasser ins Deutsche übersetzt

6 http://de.wikipedia.org/wiki/Emulsion, zuletzt gesehen am 12.03.2012

7 Ian Bogost: Alien Phenomenology: S. 13

8 Ian Bogost: Alien Phenomenology: S. 12

9 Ian Bogost: Alien Phenomenology: S. 12, wurde vom Verfasser ins Deutsche übersetzt

10 Ian Bogost: Alien Phenomenology: S. 12, wurde vom Verfasser ins Deutsche übersetzt

11 Ian Bogost: Alien Phenomenology: S. 17

12 Ian Bogost: Alien Phenomenology: S. 19

13 Ian Bogost: Alien Phenomenology: S. 20

14 Ian Bogost: Alien Phenomenology: S. 22, wurde vom Verfasser ins Deutsche übersetzt

15 http://de.wikipedia.org/wiki/Schwarzes_Loch, zuletzt gesehen am 12.03.2012

16 Phil Berardelli: Does Our Universe Live Inside a Wormhole?, http://news.sciencemag.org/sciencenow/2010/04/does-our-universe-live-inside-a-.html, zuletzt gesehen am 12.03.2012

17 Alan C. Kay: The Early History Of Smalltalk, http://propella.sakura.ne.jp/earlyHistoryST/EarlyHistoryST.html, zuletzt gesehen am 12.03.2012, wurde vom Verfasser ins Deutsche übersetzt

18 Ian Bogost: Alien Phenomenology: S. 25

19 Ian Bogost: Alien Phenomenology: S. 25, wurde vom Verfasser ins Deutsche übersetzt

20 Ian Bogost: Alien Phenomenology: S. 27

21 Ian Bogost: Alien Phenomenology: S. 28

22 Ian Bogost: Alien Phenomenology: S. 28, wurde vom Verfasser ins Deutsche übersetzt

23 http://de.wiktionary.org/wiki/Objekt, zuletzt gesehen am 12.03.2012

24 Ian Bogost: Alien Phenomenology: S. 25, wurde vom Verfasser ins Deutsche übersetzt

25 Martin Heidegger: Die Technik und die Kehre. Stuttgart, 1962, S 5.

26 A new Kind of Science, S??, wurde vom Verfasser ins Deutsche übersetzt

27 http://de.wikipedia.org/wiki/Turing-Vollst%C3%A4ndigkeit, zuletzt gesehen am 12.03.2012

28 Carl Sagan: Unser Kosmos, Ausschnitt: http://www.youtube.com/watch?v=UnURElCzGc0, zuletzt gesehen am 12.03.2012

29 Edwin Abbott Abbott: Flatland. A Romance of Many Dimensions.

30 Ian Bogost: Alien Phenomenology: S. 64

31 http://www.lyricsfreak.com/r/richard+rodgers/favorite+things_20541551.html, zuletzt gesehen am 12.03.2012

32 Ian Bogost: Alien Phenomenology: S. 38

33 Ian Bogost: Alien Phenomenology: S. 36

34 Ian Bogost: Alien Phenomenology: S. 38

35 Ian Bogost: Alien Phenomenology: S. 45, wurde vom Verfasser ins Deutsche übersetzt

36 François Blanciak: Siteless. Massachusetts, 2008, S. ix, wurde vom Verfasser ins Deutsche übersetzt

37 Ian Bogost: Alien Phenomenology: S. 93, wurde vom Verfasser ins Deutsche übersetzt

38 Ian Bogost: Alien Phenomenology: S. 100, wurde vom Verfasser ins Deutsche übersetzt

39 Vom Verfasser: http://think-jeckel.de/buch/ontology.html

40 http://llvm.org/, zuletzt gesehen am 12.03.2012

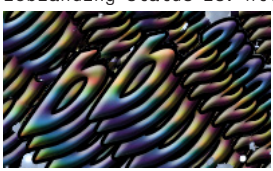
41 http://de.wikipedia.org/wiki/X86-Prozessor, zuletzt gesehen am 12.03.2012

42 http://de.wikipedia.org/wiki/ARM-Architektur, zuletzt gesehen am 12.03.2012

BILDVERZEICHNIS

stuff

Woke up a little bit late for this #LD48 but I have a direction I want to go, so let's do this! | I'm almost finished #LD48. Not only the game, but myself, too. Now I need a nice title… | Finished! WOHA! Play her: http://bureaubureau.de/article/you_are_the_hero/index.html … #LD48 I'm going to sleep now! | Lost in Skyrim Mods | My Christmas project:Writing an a* for #impactjs http://think-jeckel.de/stuff/a_stern/ .Kinda happy how fast this is and there is still room for improvement | You can click to add some more entities. Just don't click into an obstacle. Works pretty well for me until over 300 entities are reached. | Did some performance testing again and browser really got fast. Wuha. Even ie. http://think-jeckel.de/stuff/a_stern2 | Had some fun tonight, with photoshop http://think-jeckel.de/dl/bb2.gif | In other news, bureaubureau current rebranding status is: http://think-jeckel.de/dl/bb2.png . Yeah,



we are that good | Finished my js1k entry yesterday. I'm quite happy. http://bureaubureau.de/article/treasure_quest/ | So much Objects. I can't see anything anymore. | And that's it, finished my Bachelor Report right on time | Thanks to @p01 I added some animations to my js1k entry: http://data.bureaubureau.de/lab/maze/ . | All the graphical information are stored in this weird string: ~ÿÿÿÿÿÿ~ <zr$ 6LL6 6\\\\6 v6P P6v | Yeah, i'm having fun today writing code like this: double (*FP)() = (double (*)())(intptr_t)FPtr | It almost took my sanity but i finally have arrays in my little language. | This is maybe the fourth question I started to write for stackoverflow that I answered myself while typing. | Just discovered that I can answer my

own questions. As a good internet citizen I should do that, shouldn't I? | But my way was guessing, so the answer may be wrong. What to do… Better get some sleep… | That moment when you stop guessing and start knowing #woha |

sind Imagination und Kontrolle: Ihr Ästhetik liegt im abstraktem und kommt dabei dem Comics näher als dem Film. Das natürliche dar zu stellen ist absurd. Es kann nicht gewährleistet werden, das genau die eine Interpretation die der Schaffer sich erdacht hat, eingehalten wird. Der Spieler sieht, was er sehen will. Je natürlicher dies ist, desto weniger muss er sich dazu denken, aber desto begrenzter ist der Wirkungsraum. Sie sind interaktiv und da nicht jede Aktion gewollte Konsequenzen haben kann, entsteht ein Möglichkeitenraum von Fehlern. Der Spieler kontrolliert das Spiel. Der Entwickler kontrolliert das Erlebnis. Möchte er zumindest, meißtens. Kann er aber nicht. Der Spieler macht, was er will. Und darum geht es. Nicht nur das, was der Entwickler möchte. Dieser kann nur Hinweise geben. Je subtiler diese sind, desto besser. Der Spieler ist in Kontrolle, und fühlt sich nicht an die hand gepackt. | Design by Principle | Design by Substraction | What Limits? I don't belive in limits. The moment you belive in limitsm even pushing them, you have them | Zelda, The path, Dear Esther, Medroid | Die Geschichte der Dinge in Videospielen | Normally the player is an anomanlity in the game world. He is different, he can do stuff. Stuff reacts to him. Waits for him. Is there for him. What if the player is ther objekt in the game world? Just a tiny bit of the

Gerät, Stück, Bauteil, Einheit, Bestandteil, Teil | Wie alles zusammen passt: Unit -> Metaphorismn -> Carpentry -> Video Game | Heideggers Zeug: Gegenstand der zu einem Zweck dienlich ist | Wagners Stuff: Dinge die einem unbestimmten Zweck dienlich sind | Das Nichts absolut denken, nicht als Negation des Sein | Form ist

nicht verschieden von Leerheit, Leerheit ist nicht verschieden von Form | 42 | Entity -> Sprite -> Image | Be pretentious | Lonly Astronout: On a space plane. With a crew. Doing shores.

Has to go out to check/repair something. A little ruttle. You get seperated from the plane. Floating endlessly in space. Limited air to breath. Limited time in this endless space. Whats happening? | DELOCK Kabel USB 2.0-A > USBmini 5pin gew. 0,5m (http://www.amazon.de/gp/product/B0042RRNHK/ref=oh_details_o04_s00_i00?ie=UTF8&psc=1), Rocksmith Kabel (http://www.amazon.de/gp/product/B007KGISK6/ref=oh_details_o03_s00_i00?ie=UTF8&psc=1), The Quadruple Object (http://www.amazon.de/gp/product/1846947006/ref=oh_details_o02_s00_i00?ie=UTF8&psc=1), Scientific Progress Goes Boink: A Calvin and Hobbes Collection (http://www.amazon.de/gp/product/0836218787/ref=oh_details_o00_s00_i01?ie=UTF8&psc=1), Calvin und Hobbes, Band 6: Wissenschaftlicher Fortschritt macht „Boing“: BD 6 (http://www.amazon.de/gp/product/355178616X/ref=oh_details_o00_s00_i00?ie=UTF8&psc=1), iRobot Roomba 780 Staubsaug-Roboter / Funkfernbedienung / Programmierfunktion / Extra Bürstenset / Multi-Raumerkennung / HEPA ähnliche Filter / 2 Vir (http://www.amazon.de/gp/product/B005DE8EXS/ref=oh_details_o09_s00_i00?ie=UTF8&psc=1), God Hand (http://www.amazon.de/gp/product/B000KIX950/ref=oh_details_o08_s00_i00?ie=UTF8&psc=1), Land of Lisp: Learn to Program in Lisp, One Game at a Time! (http://www.amazon.de/gp/product/1593272812/ref=oh_details_o07_s00_i00?ie=UTF8&psc=1), Legend of Zelda (http://www.amazon.de/gp/product/1616550414/ref=oh_details_o06_s00_i00?ie=UTF8&psc=1), ABUS Fahrradschloss 880 / 85 Black, schwarz, 7 mm / 85 cm (http://www.amazon.de/gp/product/B001BPDRTE/ref=oh_details_o05_s00_i00?ie=UTF8&psc=1), Wovon ich rede, wenn ich vom Laufen rede (http://www.amazon.de/gp/product/3442739454/ref=oh_details_o04_s00_i00?ie=UTF8&psc=1), Ridder 463130-350 Duschvorhang Textil 180 x 200 cm Neptun blau (http://www.amazon.de/gp/product/B0039PUVPC/ref=oh_details_o03_s00_i00?ie=UTF8&psc=1), Five Great Science-Fiction Novels Set (http://www.amazon.de/gp/product/048643978X/ref=oh_details_o02_s00_i01?ie=UTF8&psc=1),

Flatland: A Romance of Many Dimensions (http://www.amazon.de/gp/product/1603863745/ref=oh_details_o02_s00_i00?ie=UTF8&psc=1), Star Wench (http://www.amazon.de/gp/product/0615779298/ref=oh_details_o01_s00_i00?ie=UTF8&psc=1), Lee BROOKLYN ONE WASH L8127145 Herren Jeanshosen/ Lang, Gr. 38/ 34 Blau (one wash 45) (http://www.amazon.de/gp/product/B002PK15UW/ref=oh_details_o00_s00_i00?ie=UTF8&psc=1) | 

to Zukan Gumi, Nazo no Mi, Gokigen Yogurt, Puccho Citrus Mix, Neru Neru Nerune, Banana Choco, Koneko Meshi, DIY Dinosaur Candy, Chelsea Soft Candy, Pokepachi, Pokemon Gummy, Anpanman Gummy, Kiichigo Gumi, Marumete Pandango, Kawarinbo, e-ma, Chou Himo Q cola / soda flavor, Naruto Kintoki Caramel, Fit's Magic Mix 7, Shige Kikkusu grape sugar / vitamin C | self aware stuff | | The Quadruple Object, Grundkurs Philosophie / Logik: BD I,  Grundkurs Philosophie / Metaphysik und Naturphie: BD II, Everyone Poops, Game Engine Architecture, The C Programming Language, Siteless: 1001 Building Forms, Die Technik und die Kehre., Finnegans Wake, Close to the Machine. Mein Leben mit dem Computer, Alien Phenomenology, or What It's Like to Be a Thing, Persuasive Games: The Expressive Power of Videogames, How to Do Things with Videogames, Rise of the Videogame Zinesters: How Freaks, Normals, Amateurs, Artists, Dreamers, Drop-outs, Queers, Housewives, and People Like You Are Taking Back | m='~ÿÿÿÿÿÿ~ <zr$ 6LL6 6\\\\6 - v6P P6v';[illegible]ock";w=c.width=innerWidth;h=c.height=innerHeight;s=8;P=w*h+h;F=S=1;f=[];g=[];e=[];c=[0,2,0,-2];onkeyup=function(b){++F?(b=b.which-38,b=P+b%2+--b%2*w,1<g[b]&&(P=b,2<g[b]&&(g[b]=2,S++))):0<s&&setTimeout(onkeyup,99,s--);for(i in e)u=P%w<e[i]%w?-1:P%w>e[i]%w?1:0,o=P<e[i]?-1:P>e[i]?1:0,e[i]+=u&1<g[e[i]+u]?u:1<g[e[i]+o*w]?o*w:0,e[i]==P&&onkeyup(e[i]=F=NaN);for(b=P;;){g[b]=2;g[b-2*w]||(g[b-w]=f[b-2*w]=1);g[b+2]||(g[b+1]=f[b+2]=1);g[b+2*w]||(g[b+w]=f[b+2*w]=1);g[b-2]||(g[b-1]=f[b-2]=1);0.01>Math.random()&P!=b&&(g[b]=3);b=1;for(i in f)if(f[i]&Math.abs(i%w-P%w)+Math.abs((i/w|0)-P/w|0)<s){b=f[i]=0;break}if(b)break;b=1*i;Math.random()<0.01*S&&e.push(b);o=4*Math.random()|0;for(i=4;i--&&!(1<g[b+c[o]+c[(o+1)%4]*w]&&(g[((b/w|0)+c[(o+1)%4]/2)*w

+b%w+c[o]/2]=2,0.1<Math.random()));o=++o%4);}a.fillRect(0,0,w,h);u=P%w-w/2/(2*s)|0;o=(P/w|0)-h/2/(2*s)|0;for(x=s;x--;)for(y=s;y--;){a.fillStyle="#fff";for(i in g)b=m.charCodeAt(x+(g[i]-1)*s),b=1==b?0:b,b>>y&1||a.fillRect(2*(i%w-u)*s+2*x,2*((i/w|0)-o)*s+2*y,2,2);a.fillStyle="#000";for(i in e)m.charCodeAt(x+24+F%2*s)>>y&1&&a.fillRect(2*(e[i]%w-u)*s+2*x,2*((e[i]/w|0)-o)*s+2*y,2,2);m.charCodeAt(x+40+F%2*s)>>y&1&&a.fillRect(2*(P%w-u)*s+2*x,2*((P/w|0)-o)*s+2*y,2,2)}top.document.title=S-1+" \u2b16"};onkeyup({}) | explode(Teil) && implode(Teil), Bits als teile, Wolfram & Turing | Oder nicht ganz so weit gehend: ooo Paradigma, Keine Vererbung. In Richtung Entität component System? Prototypen? | Warten auf godot mmo | The Humand dialect of Electronics | Programming as a philosophical practice: Flat Ontologie raus, tiny in die Mitte, Operationen zw. Units | Metal ist Musical Musik | Zwei Bücher. Oder eins + booklet. Aufbauen wie extra von alten pc spielen. Cliffhanger im ersten. Poster im zweiten. Foto in Bezug zum Apparat stellen, Vierte Dimension Schatten, who knows what they think? | the dream that run away: a dream pcisically leaves the body of the dreamer and goes on to see the night, but hurry, you only have till the sundown | i just found this folder named »autocorrect reality disbelive of unrealtistic messages« on my desktop. I bet that meant something, a thought I was having some day worthy enough to put it down. But now, they are just words. I can't connect to them.| Die verschiedenen Konzepte mittels einfachen Punkten visualisieren. Einer der Punkte ist steuerbar. Wie verhalten sich die anderen? | Boredom. I miss you. | Dissolvement of disbelive | designing design, haptic design, mold your material, physical material is not there, just input device, let it have a soul that is worth carving, let me play the song of my people | Bauarbeiter stehen immer rum. In Gruppen und machen nichts. Manchmal buddeln sie ein Loch oder bedienen eine Maschine. Einer schaut immer zu. Irgendwann, meist plötzlich, sind sie fertig, die Straße steht und kann benutzt werden. | killing provides the opposite effect? It damages you, the opponent

gets live. The goal is the be shot the most. | a game about the wringles under your eyes | What are you doing, what is happening, what makes you feel? | one button for one movement, touch, left right forward shoot | I always wanted to break one of them. Not on purpose, that should be silly. But just break some of them. See how that feels. Crunch. | Wie viel Information brauch der Spieler von seiner Spielumgebung, um damit spielen zu können? Ist es eher das informieren oder das enthalten von Information was den Moment spannend macht. Muss der Moment eindeutig definiert sein? | no one. who is one? if not me? who else could it be? Is it you? I don't know. Do you know? | where does ooo start? or better, where does it end? Is this thought a thing? IS THIS DOCUMENT A THING? where does it end. Nowhere | what are fragments? are they a thing of their own? Do they excist besides their incompleteness? Yes | I'm playing something. Killing some dudes, they probably did something bad. There is a big dude, he takes more hits than the others. But I don't care, just hit the buttons and shoot. Down he goes. Then it happens again. The screen goes black. The next level is loading. I'm looking at myself. Ask what I'm doing here. Playing some shitty game with shitty morals and shitty everything. It keeps the mind busy from other things that would probably be more stressfull. But this pause, it gets me. The void with my reflection. Something from me is there on the other side and it's looking back. Does it want to tell me something? Does it want me to stop? What does this lifeless face want. It just stares at me. Something is definetly wrong. Does it want to attack me? Should I be scared? And right in this moment the game continues, I start killing dudes again and everything is normal. Till the next loading screen. | We made the digital. It is our world. We played god with zeroes and ones. We ruled it while it was simple. We made it bigger, and complex. No one knows anymore how it works. It works on its own. | Video games like any other medium provide us with tools to express thoughts, theorethical concept and experience them in the most immersive(dare I to say this? Bad word.) way. It lets us try out things. Let's us construct theoretical worlds and try them out. What happens when I do this? What when I do this? There a no

real concequences. Just some lost time. No one took harm, everything is the same. You can not really fail in a video game world. All it does is reset to a certain state and you try again. | When I was little I had no sense I bought a pc for 800 euro. And the only thing I could play was commander keen on dos sixteen. | Gibt es Zufall? Ist alles vorherbestimmt? Ist das überhaupt ein Gegensatz? Alles besteht aus Teilchen die miteinander agieren. Bei diesem agieren entsteht Reibung. Diese Reibung ist unbestimmt. Nicht wirklicher Zufall, aber Abweichung vom Norm. Unbestimmte Abweichungen. Meinetwegen Zufall genannt, aber nicht in dem Sinne wie Zufall aufgefasst wird. Es ist nicht unbestimmt, sondern nur nicht eindeutig. Und eindeutig ist dabei eher lockerer gemeint, nicht wie in der Mathematik das Ergebnis einer Gleichung. Das Ergebnis ist nicht 1, sondern irgendwie um 1 angesiedelt. Und das im Ganzen betrachtet, entzieht der Welt jeden Zufall, aber auch jeder Vorherbestimmung. Es herrscht weder das ein noch das andere, es ist einfach. |

Bevor der werte Leser aufgrund der vorherigen Seiten völlig den Geisteszustand des Verfassers anzweifelt, ein paar erklärende Worte zu dem Begriff stuff. Stuff ist nicht wirklich definierbar. Es ist irgendwas und hat irgendwas mit irgendwas zu tun. Das steht fest. Aber welche Natur diese Bezeihung hat, ist völlig unklar. Aber das ist nicht weiter schlimm, denn es ist, was es ist. Es ist Teil des Ganzen und bieten einen völligen ungefilterten Einblick in die Thematik dieses Buches. Es ist die Welt in der das Thema lebt. Beschrieben in der rohsten Form, als einfache Liste von stuff.

Compiler QuellCode

```
#include „llvm/DerivedTypes.h“
#include „llvm/ExecutionEngine/ExecutionEngine.h“
#include „llvm/ExecutionEngine/JIT.h“
#include „llvm/IRBuilder.h“
#include „llvm/LLVMContext.h“
#include „llvm/Module.h“
#include „llvm/PassManager.h“
#include „llvm/Analysis/Verifier.h“
#include „llvm/Analysis/Passes.h“
#include „llvm/DataLayout.h“
#include „llvm/Transforms/Scalar.h“
#include „llvm/Support/TargetSelect.h“
#include <cstdio>
#include <string>
#include <map>
#include <vector>
using namespace llvm;

//
// Lexer
//

// The lexer returns tokens [0-255] (ASCII) if it is an unknown character,
otherwise one of these for known things.
enum Token {
  tok_eof = -1,

  // operations
  tok_def = -2, tok_cond = -3, tok_loop = -4,
  tok_binary = -5, tok_branch = -6, tok_assign = -7,

  // definitions
  tok_var_number = -8, tok_var_array = -9,

  // primary
  tok_identifier = -10, tok_number = -11, tok_array = -12
};

// + - * / & | ^ ~ % << >>
enum BinaryOperator {
  OP_PLUS = 1, OP_MINUS = 2, OP_MULT = 3, OP_DIVIDE = 4, OP_BITWISE_AND = 5,
  OP_BITWISE_OR = 6, OP_BITWISE_XOR = 7, OP_BITWISE_NOT = 8, OP_MODULO = 9,
```

```
  OP_LEFT_SHIFT = 10, OP_RIGHT_SHIFT = 11
};

static int IsCharBinary( std::string &Tok, int LastChar ) {
  if ( LastChar == '+' ) return OP_PLUS;
  if ( LastChar == '-' ) return OP_MINUS;
  if ( LastChar == '*' ) return OP_MULT;
  if ( LastChar == '/' ) return OP_DIVIDE;
  if ( LastChar == '&' ) return OP_BITWISE_AND;
  if ( LastChar == '|' ) return OP_BITWISE_OR;
  if ( LastChar == '^' ) return OP_BITWISE_XOR;
  if ( LastChar == '~' ) return OP_BITWISE_NOT;
  if ( LastChar == '%' ) return OP_MODULO;
  if ( Tok == „<<“ ) return OP_LEFT_SHIFT;
  if ( Tok == „>>“ ) return OP_RIGHT_SHIFT;

  return 0;
}

// == != > >= < <= && ||
enum BranchOperator {
  OP_EQUAL = 1, OP_NOT_EQUAL = 2, OP_GREATER = 3, OP_GREATER_EQUAL = 4,
  OP_LESS = 5, OP_LESS_EQUAL = 6, OP_LOGICAL_AND = 7, OP_LOGICAL_OR = 8
};

static int IsCharBranch( std::string &Tok, int LastChar ) {
  if ( Tok == „==“ ) return OP_EQUAL;
  if ( Tok == "!=" ) return OP_NOT_EQUAL;
  if ( Tok == ">=" ) return OP_GREATER_EQUAL;
  if ( Tok == "<=" ) return OP_LESS_EQUAL;
  if ( Tok == "&&" ) return OP_LOGICAL_AND;
  if ( Tok == "||" ) return OP_LOGICAL_OR;
  if ( LastChar == '>' ) return OP_GREATER;
  if ( LastChar == '<' ) return OP_LESS;

  return 0;
}

enum AssignOperator {
  OP_SET_EQUAL = 1
};

static int IsCharAssign( std::string &Tok, int LastChar ) {
  if ( LastChar == '=' ) return OP_SET_EQUAL;

  return 0;
}

static int IdentifierChar = 0;
static std::string IdentifierStr;  // Filled in if tok_identifier
static double NumVal;              // Filled in if tok_number
```

```
static int LastChar = ' ';

// gettok - Return the next token from standard input.
static int gettok() {
  // Skip any whitespace.
  while ( isspace( LastChar ) ) {
    LastChar = getchar();
  }

  // fprintf( stderr, "LastChar: %c\n", LastChar );

  // parse operation
  if ( isalpha( LastChar ) ) { // identifier: [a-zA-Z][a-zA-Z0-9]*
    IdentifierStr = LastChar;

    while ( isalnum( (LastChar = getchar()) ) ) {
      IdentifierStr += LastChar;
    }

    if ( IdentifierStr == "def" ) return tok_def;
    if ( IdentifierStr == "cond" ) return tok_cond;
    if ( IdentifierStr == "while" ) return tok_loop;
    if ( IdentifierStr == "num" ) return tok_var_number;
    if ( IdentifierStr == "array" ) return tok_var_array;

    return tok_identifier;
  }

  if ( isdigit( LastChar ) || LastChar == '.' ) {   // Number: [0-9.]+
    std::string NumStr;

    do {
      NumStr += LastChar;
      LastChar = getchar();
    } while ( isdigit( LastChar ) || LastChar == '.' );

    NumVal = strtod( NumStr.c_str(), 0 );

    return tok_number;
  }

  if ( LastChar == '\'' ) {
    NumVal = getchar();
    while ( getchar() != '\'' ) {} // eat the rest of the string
    LastChar = getchar(); // eat '
    return tok_number;
  }

  if ( LastChar == '#' ) {
    // Comment until end of line.
    do LastChar = getchar();
```

```
    while ( LastChar != EOF && LastChar != '\n' && LastChar != '\r' );

    if ( LastChar != EOF ) return gettok();
  }

  // check for multi char operators
  int NextChar = getchar();
  std::string Tok;
  Tok += LastChar; Tok += NextChar; // todo: man im dumb

  // check for + - * / & | ^ ~ % << >>
  if ( char C = IsCharBinary( Tok, LastChar ) ) {
    IdentifierChar = C;
    LastChar = getchar();
    return tok_binary;
  }

  // check for == != > >= < <= && ||
  if ( char C = IsCharBranch( Tok, LastChar ) ) {
    IdentifierChar = C;
    LastChar = getchar();
    return tok_branch;
  }

  if ( char C = IsCharAssign( Tok, LastChar ) ) {
    IdentifierChar = C;
    LastChar = getchar();
    return tok_assign;
  }

  ungetc( NextChar, stdin ); // put it back

  // Check for end of file.  Don't eat the EOF.
  if ( LastChar == EOF ) return tok_eof;

  // Otherwise, just return the character as its ascii value.
  int ThisChar = LastChar;
  LastChar = getchar();

  return ThisChar;
}


//
// Abstract Syntax Tree (aka Parse Tree)
//

enum VarTypes {
  TYPE_NUMBER = 1, TYPE_ARRAY = 2, TYPE_ARRAY_INDEX = 3
};
```

```
class ExprAST {
public:
  virtual ~ExprAST() {}
  virtual Value *Codegen() = 0;
};

class NumberExprAST : public ExprAST {
  double Val;
public:
  NumberExprAST(double val)
    : Val(val) {}
  virtual Value *Codegen();
  const double getVal() const { return Val; }
};

class ArrayExprAST : public ExprAST {
  std::vector<double> Vals;
public:
  ArrayExprAST( std::vector<double> vals )
    : Vals( vals ) {}
  virtual Value *Codegen();
};

class BinaryExprAST : public ExprAST {
  char Op;
  std::vector<ExprAST*> Args;
public:
  BinaryExprAST( char op, std::vector<ExprAST*> args )
    : Op(op), Args(args) {}
  virtual Value *Codegen();
};

class BranchExprAST : public ExprAST {
  char Op;
  std::vector<ExprAST*> Args;
public:
  BranchExprAST( char op, std::vector<ExprAST*> args )
    : Op(op), Args(args) {}
  virtual Value *Codegen();
};

class ConditionAST : public ExprAST {
  ExprAST *Condition, *Truthy, *Falsy;
public:
  ConditionAST( ExprAST *condition, ExprAST *truthy, ExprAST *falsy )
    : Condition(condition), Truthy(truthy), Falsy(falsy) {}
  virtual Value *Codegen();
};

class LoopAST : public ExprAST {
  ExprAST *Condition, *Body;
```

```
public:
  LoopAST( ExprAST *condition, ExprAST *body )
    : Condition(condition), Body(body) {}
  virtual Value *Codegen();
};

class CallExprAST : public ExprAST {
  std::string Callee;
  std::vector<ExprAST*> Args;
public:
  CallExprAST(const std::string &callee, std::vector<ExprAST*> &args)
    : Callee(callee), Args(args) {}
  virtual Value *Codegen();
};

class ParentExprAST : public ExprAST {
  std::vector<ExprAST*> Expressions;
public:
  ParentExprAST( std::vector<ExprAST*> expressions )
    : Expressions( expressions ) {}
  virtual Value *Codegen();
};

class VariableExprAST : public ExprAST {
  std::string Name;
  static const int Type = 0;
public:
  virtual const std::string &getName() const { return Name; }
  virtual const int getType() const { return Type; }
};

class NumVariableExprAST : public VariableExprAST {
  std::string Name;
  static const int Type = TYPE_NUMBER;
public:
  NumVariableExprAST( const std::string &name )
    : Name( name ) {}
  const std::string &getName() const { return Name; }
  const int getType() const { return Type; }
  virtual Value *Codegen();
};

class ArrayVariableExprAST : public VariableExprAST {
  std::string Name;
  static const int Type = TYPE_ARRAY;
public:
  ArrayVariableExprAST( const std::string &name )
    : Name( name ) {}
  const std::string &getName() const { return Name; }
  const int getType() const { return Type; }
  virtual Value *Codegen();
```

```
};

class ArrayIndexVariableExprAST : public VariableExprAST {
  std::string Name;
  ExprAST *Index;
  static const int Type = TYPE_ARRAY_INDEX;
public:
  ArrayIndexVariableExprAST( const std::string &name, ExprAST *idx )
    : Name( name ), Index( idx ) {}
  const std::string &getName() const { return Name; }
  const int getType() const { return Type; }
  ExprAST * getIndex() const { return Index; }
  virtual Value *Codegen();
};

class AssignExprAST : public ExprAST {
  char Op;
  VariableExprAST *Variable;
  ExprAST *NewValue;
public:
  AssignExprAST( char op, VariableExprAST *variable, ExprAST *newValue )
    : Op(op), Variable( variable ), NewValue( newValue ) {}
  virtual Value *Codegen();
};

class NumVarExprAST : public ExprAST {
  std::vector<std::pair<std::string, ExprAST*> > VarNames;
  ExprAST *Body;
public:
   NumVarExprAST( const std::vector<std::pair<std::string, ExprAST*> > &var-
names, ExprAST *body )
    : VarNames( varnames ), Body( body ) {}
  virtual Value *Codegen();
};

class ArrayVarExprAST : public ExprAST {
  std::vector<std::pair<std::string, NumberExprAST*> > VarNames;
  ExprAST *Body;
public:
   ArrayVarExprAST( const std::vector<std::pair<std::string, NumberExprAST*>
> &varnames, ExprAST *body )
    : VarNames( varnames ), Body( body ) {}
  virtual Value *Codegen();
};

// PrototypeAST - This class represents the "prototype" for a function,
which captures its name, and its argument names (thus implicitly the number
of arguments the function takes).
class PrototypeAST {
  std::string Name;
  std::vector<std::pair<std::string, int> > Args;
```

```
public:
    PrototypeAST(const std::string &name, const std::vector<std::pair<st-
d::string, int> > &args)
    : Name( name ), Args( args ) {}
  Function *Codegen();
  void CreateArgumentAllocas(Function *F);
};

// FunctionAST - This class represents a function definition itself.
class FunctionAST {
  PrototypeAST *Proto;
  ExprAST *Body;
public:
  FunctionAST( PrototypeAST *proto, ExprAST *body )
    : Proto( proto ), Body( body ) {}
  Function *Codegen();
};

class DefExprAST : public ExprAST {
  FunctionAST *Body;
public:
  DefExprAST( FunctionAST *body )
    : Body( body ) {}
  virtual Value *Codegen();
};

//
// Parser
//

// CurTok/getNextToken - Provide a simple token buffer.  CurTok is the cur-
rent token the parser is looking at.  getNextToken reads another token from
the lexer and updates CurTok with its results.
static int CurTok;
static int getNextToken() {
  return CurTok = gettok();
}

// Remembers variables
static std::map<std::string, int> NamedVariables;

ExprAST *Error(const char *Str) { fprintf(stderr, "Error: %s\n", Str);return
0;}
PrototypeAST *ErrorP(const char *Str) { Error(Str); return 0; }
FunctionAST *ErrorF(const char *Str) { Error(Str); return 0; }

static ExprAST *ParseExpression();

static ExprAST *ParseNumberExpr() {
  ExprAST *Result = new NumberExprAST( NumVal );
  getNextToken(); // consume the number
```

```
  return Result;
}

static ExprAST *ParseParenExpr() {
  getNextToken(); // eat (

  std::vector<ExprAST*> Expressions;

  while ( CurTok != ')' ) {
    ExprAST *V = ParseExpression();
    if ( !V ) return 0;

    Expressions.push_back( V );
  }
  getNextToken();  // eat )

  return new ParentExprAST( Expressions );
}

static VariableExprAST *ParseVariableExpr() {
  std::string Name = IdentifierStr;

  bool isArrayIndex = LastChar == '[';
  getNextToken();  // eat identifier

  if ( isArrayIndex ) {
    getNextToken(); // eat [

    ExprAST *NumExpr = ParseExpression();
    if ( NumExpr == 0 ) return 0;

    getNextToken(); // eat ]

    return new ArrayIndexVariableExprAST( Name, NumExpr );
  }
  else {
    switch ( NamedVariables[Name] ) {
      case TYPE_NUMBER: return new NumVariableExprAST( Name );
      case TYPE_ARRAY:  return new ArrayVariableExprAST( Name );
       default: fprintf( stderr, "Name: %s", Name.c_str()); Error( "Unknown
Variable" ); return 0;
    }
  }
}

static ExprAST *ParseIdentifierExpr() {
  if ( isspace( LastChar ) || LastChar == '[' || LastChar == ']' || LastChar
== ')' ) {
    return ParseVariableExpr();
  }
```

```
  std::string Name = IdentifierStr;

  getNextToken();  // eat identifier

  std::vector<ExprAST*> Args;
  getNextToken(); // eat (

  if ( CurTok != ')' ) {
    while ( 1 ) {
      ExprAST *Arg = ParseExpression();
      if ( Arg == 0 ) return 0;
      Args.push_back( Arg );

      if ( CurTok == ')' ) break;
    }
  }

  // fprintf( stderr, "Call: %s Args: %ld\n", Name.c_str(), Args.size() );

  getNextToken(); // eat )
  return new CallExprAST( Name, Args );
}

static ExprAST *ParseNumVarExpr() {
  getNextToken(); // eat num

  std::vector<std::pair<std::string, ExprAST*> > VarNames;
  std::vector<int> OldVariables;

  if ( CurTok != tok_identifier ) {
    return Error( "Expected identifier after num" );
  }

  while ( 1 ) {
    std::string Name = IdentifierStr;
    getNextToken(); // eat IdentifierStr

    ExprAST *Init = 0;

    if ( CurTok != ',' && CurTok != '(' ) {
      Init = ParseExpression();
      if ( Init == 0 ) return 0;
    }

    VarNames.push_back( std::make_pair( Name, Init ) );
    OldVariables.push_back( NamedVariables[Name] );
    NamedVariables[Name] = TYPE_NUMBER;

    if ( CurTok != ',' ) break;
    getNextToken(); // eat ,
```

```
    if ( CurTok != tok_identifier ) {
      return Error( "expected identifier list after num" );
    }
  }

  ExprAST *Body = ParseExpression();
  if ( Body == 0 ) return 0;

  // revert OldVariables
  for ( unsigned i = 0, e = VarNames.size(); i != e; ++i ) {
    NamedVariables[VarNames[i].first] = OldVariables[i];
  }

  // fprintf( stderr, "Num of Vars: %ld\n", VarNames.size() );

  return new NumVarExprAST( VarNames, Body );
}

static ExprAST *ParseArrayVarExpr() {
  getNextToken(); // eat array

  std::vector<std::pair<std::string, NumberExprAST*> > VarNames;
  std::vector<int> OldVariables;

  if ( CurTok != tok_identifier ) {
    return Error( "Expected identifier after var" );
  }

  while ( 1 ) {
    std::string Name = IdentifierStr;
    getNextToken(); // eat IdentifierStr

    if ( CurTok != tok_number ) {
      Error( "Arrays need the length as a number" );
      return 0;
    }

    NumberExprAST *Length = (NumberExprAST*) ParseNumberExpr();
    if ( Length == 0 ) return 0;

    VarNames.push_back( std::make_pair( Name, Length ) );
    OldVariables.push_back( NamedVariables[Name] );
    NamedVariables[Name] = TYPE_ARRAY;

    if ( CurTok != ',' ) break;
    getNextToken(); // eat ,

    if ( CurTok != tok_identifier ) {
      return Error( "expected identifier list after var" );
    }
```

```
  }

  ExprAST *Body = ParseExpression();
  if ( Body == 0 ) return 0;

  // revert OldVariables
  for ( unsigned i = 0, e = VarNames.size(); i != e; ++i ) {
    NamedVariables[VarNames[i].first] = OldVariables[i];
  }

  // fprintf( stderr, "Num of Arrays: %ld\n", VarNames.size() );

  return new ArrayVarExprAST( VarNames, Body );
}

static ExprAST* ParsePrimary() {
  // fprintf( stderr, "CurTok: %d or %c\n", CurTok, CurTok );
  switch ( CurTok ) {
    default: return Error( "unknown token when expecting an expression" );
    case tok_identifier: return ParseIdentifierExpr();
    case tok_number:     return ParseNumberExpr();
    case '(':            return ParseParenExpr();
    case tok_var_number: return ParseNumVarExpr();
    case tok_var_array:  return ParseArrayVarExpr();
  }
}

static ExprAST *ParseBinary() {
  int BinOp = IdentifierChar;
  getNextToken(); // eat op

  std::vector<ExprAST*> Args;

  while ( CurTok != ')' ) {
    ExprAST *Arg = ParsePrimary();
    if ( !Arg ) return 0;

    Args.push_back( Arg );

    if ( CurTok == ')' ) break;
  }

  // fprintf( stderr, "Name: %d Args: %ld\n", BinOp, Args.size() );

  if ( Args.size() < 2 ) {
    Error( "BinOp needs at least 2 arguments" );
    return 0;
  }

  return new BinaryExprAST( BinOp, Args );
}
```

```
static ExprAST *ParseBranch() {
  int BranchOp = IdentifierChar;

  // eat op
  getNextToken();

  std::vector<ExprAST*> Args;

  while ( CurTok != ')' ) {
    ExprAST *Arg = ParsePrimary();
    if ( !Arg ) return 0;

    Args.push_back( Arg );

    if ( CurTok == ')' ) break;
  }

  if ( Args.size() < 2 ) {
    Error( "BranchOp needs at least 2 arguments" );
  }

  // fprintf( stderr, "BranchOp: %d Args: %ld\n", BranchOp, Args.size() );

  return new BranchExprAST( BranchOp, Args );
}

static PrototypeAST *ParsePrototype() {
  if ( CurTok != tok_identifier ) {
    return ErrorP("Expected function name in prototype");
  }

  std::string FnName = IdentifierStr;
  std::vector<std::pair<std::string, int> > ArgNames;

  getNextToken(); // eat identifier
  getNextToken(); // eat (

  while ( CurTok != ')' ) {
    int ParaType = 0;

    switch ( CurTok ) {
      case tok_var_number:  ParaType = TYPE_NUMBER; break;
      case tok_var_array:   ParaType = TYPE_ARRAY; break;
      default: Error( "Unknow Parameter Type" ); return 0;
    }

    getNextToken(); //eat type

    ArgNames.push_back( std::make_pair( IdentifierStr, ParaType ) );
    NamedVariables[IdentifierStr] = ParaType;
```

```
    getNextToken(); // eat identifier

    if ( CurTok != ',' ) break;
    getNextToken(); // eat ,
  }

  if ( CurTok != ')' ) {
    return ErrorP( "Expected ')' in prototype" );
  }

  getNextToken();  // eat )

  // fprintf( stderr, "Parsed a prototype: %s Arg Length: %ld \n", FnNa-
me.c_str(), ArgNames.size() );

  return new PrototypeAST( FnName, ArgNames );
}

static FunctionAST *ParseDefinition() {
  getNextToken();  // eat def

  PrototypeAST *Proto = ParsePrototype();

  if ( Proto == 0 ) return 0;

  if ( ExprAST *E = ParseExpression() ) {
    return new FunctionAST( Proto, E );
  }

  return 0;
}

static ExprAST *ParseCondition() {
  getNextToken();  // eat cond

  ExprAST *Condition = ParseExpression();
  if ( Condition == 0 ) return 0;
  if ( CurTok == ')' ) getNextToken();

  ExprAST *Truthy = ParseExpression();
  if ( Truthy == 0 ) return 0;
  if ( CurTok == ')' ) getNextToken();

  ExprAST *Falsy = ParseExpression();
  if ( Falsy == 0 ) return 0;

  return new ConditionAST( Condition, Truthy, Falsy );
}

static ExprAST *ParseLoop() {
```

```
  getNextToken(); // eat while

  ExprAST *Condition = ParseExpression();
  if ( Condition == 0 ) return 0;
  if ( CurTok == ')' ) getNextToken();

  ExprAST *Body = ParseExpression();
  if ( Body == 0 ) return 0;

  return new LoopAST( Condition, Body );
}

static ExprAST *ParseAssign() {
  int AssignOp = IdentifierChar;
  getNextToken(); // eat assign

  if ( CurTok != tok_identifier ) {
    return Error( "First Parameter must be an identifier" );
  }

  VariableExprAST *Variable = ParseVariableExpr();
  ExprAST *NewValue = ParseExpression();

  if ( CurTok != ')' ) {
    return Error( "Assign operator takes only one value" );
  }

  // fprintf( stderr, "AssignOp: %d VarName: %s\n", AssignOp, Variable->get-
Name().c_str() );

  return new AssignExprAST( AssignOp, Variable, NewValue );
}

static ExprAST *ParseExpression() {
  // fprintf( stderr, "CurTok: %d or %c\n", CurTok, CurTok );
  switch ( CurTok ) {
    case '('         :     return ParseParenExpr();
    case tok_number:     return ParseNumberExpr();
    case tok_binary:     return ParseBinary();
    case tok_branch:     return ParseBranch();
    case tok_cond:       return ParseCondition();
    case tok_loop:       return ParseLoop();
    case tok_identifier: return ParseIdentifierExpr();
    case tok_assign:     return ParseAssign();
    case tok_var_number: return ParseNumVarExpr();
    case tok_var_array:  return ParseArrayVarExpr();
    default: return Error( "Not a valid Expression Type" );
  }
}

static FunctionAST *ParseTopLevelExpr() {
```

```
  if ( ExprAST *E = ParseExpression() ) {
    // Make an anonymous proto.
     PrototypeAST *Proto = new PrototypeAST( "", std::vector<std::pair<st-
d::string, int> >() );
    return new FunctionAST( Proto, E );
  }

  return 0;
}

//
// Code Generation
//

static Module *TheModule;
static IRBuilder<> Builder( getGlobalContext() );
static std::map<std::string, AllocaInst*> NamedValues;

// Helper

Value *ErrorV( const char *Str ) { Error(Str); return 0; }

// Create an alloca instruction in the entry block of the function.  This
is used for mutable variables etc.
static AllocaInst *CreateEntryBlockAlloca( Function *TheFunction, const
std::pair<std::string, int> &Var, int Length = 0 ) {
   IRBuilder<> TmpB( &TheFunction->getEntryBlock(), TheFunction->getEntry-
Block().begin() );

  Type *T;

  switch ( Var.second ) {
    case TYPE_NUMBER: T = Type::getDoubleTy( getGlobalContext() ); break;
    case TYPE_ARRAY:
      if ( Length ) {
       T = ArrayType::get( Type::getDoubleTy( getGlobalContext() ), Length );
      }
      else {
          T = PointerType::getUnqual( Type::getDoubleTy( getGlobalContext()
) );
      }

      break;
    case TYPE_ARRAY_INDEX: break; // todo?
    default: Error( "No known Type" ); return 0;
  }

  return TmpB.CreateAlloca( T, 0, Var.first.c_str() );
}

// Codegen
```

```
Value *NumberExprAST::Codegen() {
  return ConstantFP::get( getGlobalContext(), APFloat( Val ) );
}

Value *ArrayVariableExprAST::Codegen() {
    return  Builder.CreateBitCast(  NamedValues[Name],  Type::getDoublePtrTy(
getGlobalContext() ) );
}

Value *BinaryExprAST::Codegen() {
  std::vector<Value*> ArgValues;

  for ( unsigned i = 0, e = Args.size(); i != e; ++i ) {
    ArgValues.push_back( Args[i]->Codegen() );
    if ( ArgValues.back() == 0 ) return 0;
  }

  // bitwise operation only work with int types
  Value *CurrentValue;
  Value *TempValue;

  if ( Op == OP_BITWISE_AND || Op == OP_BITWISE_OR || Op == OP_BITWISE_XOR
|| Op == OP_RIGHT_SHIFT || Op == OP_LEFT_SHIFT ) {
    CurrentValue = Builder.CreateFPToUI( ArgValues[0], Type::getInt64Ty( get-
GlobalContext() ), "inttmp" );
  }
  else CurrentValue = ArgValues[0];

  for ( unsigned i = 1, e = ArgValues.size(); i != e; ++i ) {
    switch ( Op ) {
      case OP_PLUS:   CurrentValue = Builder.CreateFAdd( CurrentValue, Arg-
Values[i], "addtmp" ); break;
      case OP_MINUS:  CurrentValue = Builder.CreateFSub( CurrentValue, Arg-
Values[i], "subtmp" ); break;
      case OP_MULT:   CurrentValue = Builder.CreateFMul( CurrentValue, Arg-
Values[i], "multmp" ); break;
      case OP_DIVIDE: CurrentValue = Builder.CreateFDiv( CurrentValue, Arg-
Values[i], "divtmp" ); break;
      case OP_MODULO: CurrentValue = Builder.CreateFRem( CurrentValue, Arg-
Values[i], "remtmp" ); break;
      // convert for bitwise to int
      case OP_BITWISE_AND:
          TempValue = Builder.CreateFPToUI( ArgValues[i], Type::getInt64Ty(
getGlobalContext() ), "inttmp" );
        CurrentValue = Builder.CreateAnd( CurrentValue, TempValue, "andtmp"
); break;
      case OP_BITWISE_OR:
          TempValue = Builder.CreateFPToUI( ArgValues[i], Type::getInt64Ty(
getGlobalContext() ), "inttmp" );
         CurrentValue = Builder.CreateOr( CurrentValue, TempValue, "ortmp"
```

```
); break;
      case OP_BITWISE_XOR:
          TempValue = Builder.CreateFPToUI( ArgValues[i], Type::getInt64Ty(
getGlobalContext() ), "inttmp" );
         CurrentValue = Builder.CreateXor( CurrentValue, TempValue, "xortmp"
); break;
      case OP_RIGHT_SHIFT:
          TempValue = Builder.CreateFPToUI( ArgValues[i], Type::getInt64Ty(
getGlobalContext() ), "inttmp" );
        CurrentValue = Builder.CreateLShr( CurrentValue, TempValue, "rsttmp"
); break;
      case OP_LEFT_SHIFT:
          TempValue = Builder.CreateFPToUI( ArgValues[i], Type::getInt64Ty(
getGlobalContext() ), "inttmp" );
         CurrentValue = Builder.CreateShl( CurrentValue, TempValue, "lsttmp"
); break;
      // case '~': CurrentValue = Builder.CreateFAdd( CurrentValue, ArgValu-
es[i], "addtmp" ); break;
        default: return ErrorV( "invalid binary operator" );
    }
  }

  if ( Op == OP_BITWISE_AND || Op == OP_BITWISE_OR || Op == OP_BITWISE_XOR
|| Op == OP_RIGHT_SHIFT || Op == OP_LEFT_SHIFT ) {
     CurrentValue = Builder.CreateUIToFP( CurrentValue, Type::getDoubleTy(
getGlobalContext() ), "flttmp" );
  }

  return CurrentValue;
}

Value *BranchExprAST::Codegen() {
  std::vector<Value*> ArgValues;

  for ( unsigned i = 0, e = Args.size(); i != e; ++i ) {
    ArgValues.push_back( Args[i]->Codegen() );
    if ( ArgValues.back() == 0 ) return 0;
  }

  Value *TempValue;
  Value *CurrentValue;

  for ( unsigned i = 1, e = ArgValues.size(); i != e; ++i ) {
    switch ( Op ) {
      case OP_LESS:          TempValue = Builder.CreateFCmpULT( ArgValues[0],
ArgValues[i], "lsstmp" ); break;
     case OP_LESS_EQUAL:     TempValue = Builder.CreateFCmpULE( ArgValues[0],
ArgValues[i], "lsetmp" ); break;
     case OP_GREATER:        TempValue = Builder.CreateFCmpUGT( ArgValues[0],
ArgValues[i], "grttmp" ); break;
       case OP_GREATER_EQUAL:   TempValue = Builder.CreateFCmpUGE( ArgValu-
```

```
es[0], ArgValues[i], "gretmp" ); break;
      case OP_EQUAL:          TempValue = Builder.CreateFCmpUEQ( ArgValues[0],
ArgValues[i], "eqltmp" ); break;
      case OP_NOT_EQUAL:      TempValue = Builder.CreateFCmpUNE( ArgValues[0],
ArgValues[i], "neqtmp" ); break;
       // case OP_LOGICAL_AND:
       // case OP_LOGICAL_OR:
       default: return ErrorV( "invalid branch operator" );
     }

     if ( i == 1 ) CurrentValue = TempValue;
     else CurrentValue = Builder.CreateAnd( CurrentValue, TempValue, "cmptmp"
);
   }

    return Builder.CreateUIToFP( CurrentValue, Type::getDoubleTy( getGlobal-
Context() ), "booltmp" );
}

Value *CallExprAST::Codegen() {
  // check if function ref
  Function *CalleeF = TheModule->getFunction( Callee );
  if ( CalleeF == 0 ) {
    // if not, check for varaible ref in NamedValues
    Value *V = NamedValues[Callee];
    return V ? V : ErrorV( "Variable not found" );
  }

  if ( CalleeF->arg_size() != Args.size() ) {
    return ErrorV( "Incorrect # arguments passed" );
  }

  std::vector<Value*> ArgsV;
  for ( unsigned i = 0, e = Args.size(); i != e; ++i ) {
    ArgsV.push_back( Args[i]->Codegen() );
    if ( ArgsV.back() == 0 ) return 0;
  }

  return Builder.CreateCall( CalleeF, ArgsV, "calltmp" );
}

void PrototypeAST::CreateArgumentAllocas( Function *F ) {
  Function::arg_iterator AI = F->arg_begin();
  for ( unsigned Idx = 0, e = Args.size(); Idx != e; ++Idx, ++AI ) {
    AllocaInst *Alloca = CreateEntryBlockAlloca( F, Args[Idx] );
    Builder.CreateStore( AI, Alloca );
    NamedValues[Args[Idx].first] = Alloca;
  }
}

Function *PrototypeAST::Codegen() {
```

```
  std::vector<Type*> ArgTypes;

  for ( unsigned i = 0, e = Args.size(); i != e; ++i ) {
    Type *T;

    switch ( Args[i].second ) {
      case TYPE_NUMBER: T = Type::getDoubleTy( getGlobalContext() ); break;
       case TYPE_ARRAY: T = PointerType::getUnqual( Type::getDoubleTy( get-
GlobalContext() ) ); break;
       case TYPE_ARRAY_INDEX: T = Type::getDoubleTy( getGlobalContext() );
break;
      default: Error( "Unknown Type" ); return 0;
    }

    ArgTypes.push_back( T );
  }

   FunctionType *FT = FunctionType::get( Type::getDoubleTy( getGlobalCont-
ext() ), ArgTypes, false);

  Function *F = Function::Create( FT, Function::ExternalLinkage, Name, The-
Module );

  if ( F->getName() != Name ) {
    F->eraseFromParent();
    F = TheModule->getFunction( Name );

    if ( !F->empty() ) {
      Error( "redifinition of function" );
      return 0;
    }

    if ( F->arg_size() != Args.size() ) {
      ErrorF( "redifinition of function with differrent # args" );
      return 0;
    }
  }

  unsigned Idx = 0;
  for ( Function::arg_iterator AI = F->arg_begin(); Idx != Args.size(); ++AI,
++Idx ) {
    AI->setName( Args[Idx].first );
  }

  return F;
}

Function *FunctionAST::Codegen() {
  NamedValues.clear();

  Function *TheFunction = Proto->Codegen();
```

```
  if ( TheFunction == 0 ) return 0;

  BasicBlock *BB = BasicBlock::Create( getGlobalContext(), "entry", TheFun-
ction );
  Builder.SetInsertPoint( BB );

  Proto->CreateArgumentAllocas( TheFunction );

  if ( Value *RetVal = Body->Codegen() ) {
    Builder.CreateRet( RetVal );
    verifyFunction( *TheFunction );

    return TheFunction;
  }

  TheFunction->eraseFromParent();
  ErrorF( "ErrorBody" );
  return 0;
}

Value *ConditionAST::Codegen() {
  Value *CondV = Condition->Codegen();
  if ( CondV == 0 ) return 0;

  CondV = Builder.CreateFCmpONE( CondV, ConstantFP::get( getGlobalContext(),
APFloat( 0.0 ) ), "cndtmp" );

  Function *TheFunction = Builder.GetInsertBlock()->getParent();

  BasicBlock *ThenBB = BasicBlock::Create( getGlobalContext(), "then", The-
Function );
  BasicBlock *ElseBB = BasicBlock::Create( getGlobalContext(), "else" );
  BasicBlock *MergeBB = BasicBlock::Create( getGlobalContext(), "cnttmp" );

  Builder.CreateCondBr( CondV, ThenBB, ElseBB );
  Builder.SetInsertPoint( ThenBB );

  Value *TruthyV = Truthy->Codegen();
  if ( TruthyV == 0 ) return 0;

  Builder.CreateBr( MergeBB );
  ThenBB = Builder.GetInsertBlock();

  TheFunction->getBasicBlockList().push_back( ElseBB );
  Builder.SetInsertPoint( ElseBB );

  Value *FalsyV = Falsy->Codegen();
  if ( FalsyV == 0 ) return 0;

  Builder.CreateBr( MergeBB );
  ElseBB = Builder.GetInsertBlock();
```

```
  TheFunction->getBasicBlockList().push_back( MergeBB );
  Builder.SetInsertPoint( MergeBB );
  PHINode *PN = Builder.CreatePHI( Type::getDoubleTy( getGlobalContext() ),
2, "iftmp" );

  PN->addIncoming( TruthyV, ThenBB );
  PN->addIncoming( FalsyV, ElseBB );

  return PN;
}

Value *LoopAST::Codegen() {
  Function *TheFunction = Builder.GetInsertBlock()->getParent();
  BasicBlock *LoopBB = BasicBlock::Create( getGlobalContext(), "loop", The-
Function );

  Builder.CreateBr( LoopBB );
  Builder.SetInsertPoint( LoopBB );

  Value *B = Body->Codegen();
  if ( B == 0 ) return 0;

  Value *ConditionV = Condition->Codegen();
  if ( ConditionV == 0 ) return 0;

  ConditionV = Builder.CreateFCmpONE( ConditionV, ConstantFP::get( getGlo-
balContext(), APFloat( 0.0 ) ), "loopcond" );

  BasicBlock *LoopEndBB = Builder.GetInsertBlock();
  BasicBlock *AfterBB = BasicBlock::Create( getGlobalContext(), "afterloop",
TheFunction );

  Builder.CreateCondBr( ConditionV, LoopBB, AfterBB );
  Builder.SetInsertPoint( AfterBB );

  // return 0.0
  return Constant::getNullValue(Type::getDoubleTy(getGlobalContext()));
}

Value *ParentExprAST::Codegen() {
  std::vector<Value*> ExpValues;

  for ( unsigned i = 0, e = Expressions.size(); i != e; ++i ) {
    ExpValues.push_back( Expressions[i]->Codegen() );
    if ( ExpValues.back() == 0 ) return 0;
  }

  return ExpValues.back();
}
```

```
Value *NumVariableExprAST::Codegen() {
  Value *V = NamedValues[Name];
  if ( V == 0 ) {
    fprintf( stderr, "Name: %s\n", Name.c_str() );
    return ErrorV( "Unknown Variable Name" );
  }

  return Builder.CreateLoad( V, Name.c_str() );
}

Value *ArrayIndexVariableExprAST::Codegen() {
  Value *V = NamedValues[Name];
  if ( V == 0 ) {
    fprintf( stderr, "Name: %s\n", Name.c_str() );
    return ErrorV( "Unknown Variable Name" );
  }

  Value *IndexV = Index->Codegen();
  if ( IndexV == 0 ) return 0;

   Value *IntV = Builder.CreateFPToUI( IndexV, Type::getInt32Ty( getGlobal-
Context() ) );

  std::vector<Value*> vect;
  if ( NamedValues[Name]->getAllocatedType()->isArrayTy() ) {
     vect.push_back( ConstantInt::get( Type::getInt32Ty( getGlobalContext()
), 0 ) );
  }
  vect.push_back( IntV );

  if ( !NamedValues[Name]->getAllocatedType()->isArrayTy() ) {
    V = Builder.CreateLoad( V );
  }

   Value *VV = GetElementPtrInst::CreateInBounds( V, vect, "arr", Builder.
GetInsertBlock() );

  return Builder.CreateLoad( VV, Name.c_str() );
}

Value *AssignExprAST::Codegen() {
  Value *Val = NewValue->Codegen();
  if ( Val == 0 ) return 0;

  Value *VariableV;

  switch ( Variable->getType() ) {
    case TYPE_NUMBER: VariableV = NamedValues[Variable->getName()]; break;
    case TYPE_ARRAY: {
      VariableV = NamedValues[Variable->getName()];
```

```
      Val = Val->stripPointerCasts();
      Val = Builder.CreateLoad( Val );

      break;
    }
    case TYPE_ARRAY_INDEX: {
     ArrayIndexVariableExprAST *V = (ArrayIndexVariableExprAST *) Variable;

      Value *IndexV = V->getIndex()->Codegen();
      if ( IndexV == 0 ) return 0;

       Value *IntV = Builder.CreateFPToUI( IndexV, Type::getInt32Ty( getGlo-
balContext() ) );

      std::vector<Value*> vect;
       vect.push_back( ConstantInt::get(Type::getInt32Ty( getGlobalContext()
), 0 ) );
      vect.push_back( IntV );

        VariableV = GetElementPtrInst::CreateInBounds( NamedValues[Variab-
le->getName()], vect, "arr", Builder.GetInsertBlock() );
      break;
    }
    default: Error( "Unknown Variable Type" ); return 0;
  }

  if ( VariableV == 0 ) {
    fprintf( stderr, "Name: %s\n", Variable->getName().c_str() );
    return ErrorV( "Error assigning Variable" );
  }

  VariableV->dump();
  Val->dump();

  Builder.CreateStore( Val, VariableV );
  return Val;
}

Value *NumVarExprAST::Codegen() {
  std::vector<AllocaInst *> OldBindings;

  Function *TheFunction = Builder.GetInsertBlock()->getParent();

  for ( unsigned i = 0, e = VarNames.size(); i != e; ++i ) {
    const std::string &VarName = VarNames[i].first;
    ExprAST *Init = VarNames[i].second;

    Value *InitVal;
    if ( Init ) {
      InitVal = Init->Codegen();
      if ( InitVal == 0 ) return 0;
```

```
    }
    else {
      InitVal = ConstantFP::get( getGlobalContext(), APFloat( 0.0 ) );
    }

    std::pair<std::string, int> Var( VarName, TYPE_NUMBER );
    AllocaInst *Alloca = CreateEntryBlockAlloca( TheFunction, Var );
    Builder.CreateStore( InitVal, Alloca );

    OldBindings.push_back( NamedValues[VarName] );
    NamedValues[VarName] = Alloca;
  }

  Value *BodyV = Body->Codegen();
  if ( BodyV == 0 ) return 0;

  // revert scope
  for ( unsigned i = 0, e = VarNames.size(); i != e; ++i ) {
    NamedValues[VarNames[i].first] = OldBindings[i];
  }

  return BodyV;
}

Value *ArrayVarExprAST::Codegen() {
  std::vector<AllocaInst *> OldBindings;

  Function *TheFunction = Builder.GetInsertBlock()->getParent();

  for ( unsigned i = 0, e = VarNames.size(); i != e; ++i ) {
    const std::string &VarName = VarNames[i].first;

    int Length = (int) VarNames[i].second->getVal();

    std::pair<std::string, int> Var( VarName, TYPE_ARRAY );
    AllocaInst *Alloca = CreateEntryBlockAlloca( TheFunction, Var, Length );

    // Builder.CreateStore( ArrayV, Alloca );
    OldBindings.push_back( NamedValues[VarName] );
    NamedValues[VarName] = Alloca;
  }

  Value *BodyV = Body->Codegen();
  if ( BodyV == 0 ) return 0;

  // revert scope
  for ( unsigned i = 0, e = VarNames.size(); i != e; ++i ) {
    NamedValues[VarNames[i].first] = OldBindings[i];
  }

  return BodyV;
```

```
}

//
// Top-Level parsing and JIT Driver
//

static ExecutionEngine *TheExecutionEngine;

static void HandleDefinition() {
  if ( FunctionAST *F = ParseDefinition() ) {
    if ( Function *LF = F->Codegen() ) {
      fprintf( stderr, "Parsed a function definition.\n" );
      LF->dump();
    }
    else {
      Error( "Generate Function Code" );
    }
  }
  else {
    getNextToken();
  }
}

static void HandleTopLevelExpression() {
  if ( FunctionAST *F = ParseTopLevelExpr() ) {
    if ( Function *LF = F->Codegen() ) {
      fprintf( stderr, "Parsed a top level expression \n" );
      LF->dump();

      void *FPtr = TheExecutionEngine->getPointerToFunction( LF );

      // behold the probably most fucked up line I ever wrote
      // it casts the function to the right type: double
      // we only want doubles
      double (*FP)() = (double (*)())(intptr_t)FPtr;
      fprintf( stderr, "Evaluated to %f\n", FP() );
    }
  }
  else {
    getNextToken();
  }
}

//
// Main
//

static void MainLoop() {
  while ( 1 ) {
    fprintf( stderr, "ready> " );
```

```
    getNextToken(); // eat (

    // fprintf( stderr, "CurTok: %d or %c\n", CurTok, CurTok );

    switch ( CurTok ) {
      case '(':     break;
      case tok_eof: return;
      case tok_def: HandleDefinition(); break;
      case ')':     getNextToken(); break; // eat )
      default:      HandleTopLevelExpression(); break;
    }
  }
}

//
// Main
//

int main() {
  InitializeNativeTarget();
  LLVMContext &Context = getGlobalContext();

  fprintf( stderr, "ready> " );
  getNextToken();

  TheModule = new Module( "part jit", Context );

  std::string ErrStr;
   TheExecutionEngine  =  EngineBuilder(  TheModule  ).setErrorStr(  &ErrStr
).create();
  if ( !TheExecutionEngine ) {
    fprintf( stderr, "Could not create ExecutionEngine: %s\n", ErrStr.c_str()
);
    exit( 1 );
  }

  MainLoop();
  TheModule->dump();

  return 0;
}
```