The University of New South Wales

School of Computer Science and Engineering

Introducing *Jolog*: an Object-Oriented Cognitive Robotics Language

Timothy Cerexhe
Bachelor of Computer Science (Honours)

Supervisor: A/Prof. Maurice Pagnucco
Assessor: Prof. Claude Sammut

Submission Date: October 2009

Abstract

The cognitive, robotics language *Indigolog* provides a powerful notation for producing reasoning agents in complex, dynamic domains. However it lacks certain features that would make it more accessible, particularly in regard to syntax and program structure. The aim of this thesis is to produce a new object-oriented formulation of *Indigolog*, to develop a compiler for this language, and to ensure that it is faithful to the original specification and the underlying formal theory – the Situation Calculus. This report endeavours to document the development, the research necessary to produce it and the ultimate benefits that it can provide.

Acknowledgements

I have learnt more this year than I can justifiably cram into this report; some of it is even computing-related. I extend a general but heartfelt appreciation to all who assisted in this process or helped ease the pain.

To my friends and colleagues (you know who you are!): for feeding me, for reminding me of things I should not have forgotten (or had so-far failed to learn), for vigilantly turning the lab air-conditioner back on at three in the morning to keep the air from going stale and for driving me home when it did. I would not technically have survived this year in one piece without your collective efforts and presence. I also mention the members of the CSE Revue band for giving me an opportunity to 'rock out' when I wasn't working on this thesis: the thrill of counting my remaining undergraduate career in hours is quite inexplicable, but the chance to play keyboard half-naked with my knee to almost 700 people is pretty cool too.

I also acknowledge my amazing family for getting me this far, showering me with love and small comforts and always encouraging me on to bigger and better things. Whatever efforts are reflected by this thesis are due to a foundation that they have worked hard to foster and I am deeply grateful. I especially acknowledge my father as an inspiring role model and mentor – both personally and professionally – who at short notice valiantly dedicated his background in everything – except Computer Science – to read through this tome and still make useful comments.

Finally I want to thank my AWESOME supervisor, 'Morri', for solidifying 'Computer Science' for me at the start of my undergraduate career and for supporting my progress through to the end of it. With any luck I will continue to remember these lessons long after venturing out into the "Real World" ...

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Automated problem solving is one of the motivating forces in the field of Artificial Intelligence. It is an attractive idea to simply provide a system with a problem specification and wait for it to independently divine a solution via general purpose methods. However, the reality is that general systems cannot naively handle the computational complexity of difficult problems.

There are many approaches for producing intelligent software in this climate, ranging from abstract to very tailored solutions. *Golog* – a language based on first-order logic – is of the former category, designed specifically for reasoning in diverse, dynamic domains. *Golog* addresses computational complexity by incorporating domain-specific knowledge into otherwise general systems, a technique shared by cognitive robotics languages.

However, while *Golog* offers many nice features – including sophisticated reasoning – all existing flavours lack appropriate abstraction of multiple agents, extensive communication protocols or convenient debugging support.

The aim of this thesis is to produce a new variant of *Golog* that addresses these concerns so that the program can increase development efficiency and accurately reflect more complicated scenarios in our target iCinema applications. We intend this new version of *Golog* to exhibit the following desired features:

- an 'environment' thread for maintaining global fluents and spawning autonomous agents as child threads
- convenient agent communication protocols as language primitives
- groomed syntax and compiler error messages
- improved language efficiency via a dedicated, object-oriented virtual machine[1]

[1] We contrast this with an 'interpreted interpreter' as discussed later.

This should result in a convenient way of producing diverse communities of agents, capable of sophisticated and responsive interaction with each other and the external world. This is particularly favourable for our target domain of the UNSW iCinema – a dynamic environment where several agents have to interact in real time with multiple humans.

The final program will then be evaluated in terms of the expressive power and development speed of a series of test programs including:

- Canonical examples:
  - *Golog*-elevator
  - Wumpus world
  - Grid world
- Call-centre (for communication)
- Logical inference examples
- Travel-agent bot ('semantic web'-inspired)

Note that this thesis is targeted at the UNSW iCinema's 'Scenario' project. Accordingly, the existing *Golog* code produced over the 2008-2009 summer will also be ported for direct comparison with previous *Golog* implementations.

Chapter 2

Background

2.1 Situation Calculus

The Situation Calculus is a dialect of first-order logic that allows us to reason about actions in dynamic domains. This is achieved by augmenting standard first-order logic with several additional constructs:

- *Situations* – the history of 'actions' that have occurred and the *current* state of the universe[1].
- *Initial situation* – s_0, the initial state of the universe before any actions have occured.
- *Fluents* – situation-dependent[2] predicates representing properties of the world. Fluents are usually distinguished from other constructs by their form; they accept a situation as the final argument: $f(\ldots, s)$
- *Actions* – named terms with preconditions, used for affecting the current situation.
- *Effect Axioms*[3] – axioms that define the result of performing an action; how an action modifies fluent values and accordingly transitions one situation to another.
- *do function* – $do(\alpha[s], s) \rightarrow s'$ which returns the situation that results from preforming action α in situation[4] s.

[1] That is, a snapshot of where we are and how we got there, in contrast to *Reiter*'s definition where a situation is precisely a sequence of actions and should be considered distinct from any notion of state.

[2] Technically fluents are either *relational* (truth depends on situation) or *functional* (value depends on situation). Note that a relational fluent can be seen as a special case of functional fluent with a Boolean 'type', so we will henceforth only consider functional fluents.

[3] Note that these are usually replaced with *Successor-State Axioms* – effect axioms that exploit the 'common-sense principle of inertia' – to solve the *Frame Problem*.

[4] The notation $\alpha[s]$ is used to indicate that the action *may* have situation dependent arguments.

- *Do relation* – $Do(\delta, s, s')$ which maps the effects of a program δ in a situation s to a new situation s'.

The result of these additions is a framework for describing a current situation s' as a sequence of actions $\alpha_0 \ldots \alpha_n$ acting on some initial situation s_0:

$$s' = do(\alpha_n, do(\alpha_{n-1}, do(\ldots, do(\alpha_0, s_0))))$$

Further, special axioms allow us to describe the preconditions for and effects of these actions on fluents – situation-dependent predicates whose truth-value changes as the result of actions. We can then use these constructs – along with external 'exogenous' actions – to reason about the effects of actions in dynamic domains. That is, to plan a sequence of actions to achieve some goal situation.

Note that a formal description of the effect axioms requires the status of *every* fluent to be explicitly detailed, otherwise the truth value is not logically guaranteed to be known.

An example of this is the action *drop*. Performing this action modifies the world (situation, since it is a sequence of actions that is extended by virtue of doing an action) and we cannot know whether it has affected something else (such as whether the thing we dropped is now broken) unless the resulting value of this fluent is provided. This is known as the frame problem. A popular solution is to adopt the 'common-sense principle of inertia'. That is, in first order logic everything needs to be explicit, so an effect axiom has to identify what *doesn't* change as well. A successor-state axiom is distinguished from an effect axiom since it allows us to 'leave out' this redundant information. This allows us to assume that a fluent remains unchanged after an action if the successor-state axiom does not mention it. That is, our drop action should state whether the object actually is broken, but since it says nothing regarding other fluents (like the colour of the object) we can safely leave their values alone.

This is not the only solution to the frame problem, nor is it necessarily the best, since it relies on an additional assumption that isn't always valid (in physical robotics for example). Fortunately though, this particular method turns out to be a natural way of expressing the effects of actions in cases where a full description is unnecessary, such as in environments with well-defined mechanics like the UNSW iCinema.

In addition to the *do* function for linking situations by executing actions, we also have a *Do* relation for linking situations by executing *programs* (note capitalisation). Specifically, executing program δ in situation s can terminate in s'. Note that this is a relation (as opposed to a function, which is single-valued) because the program may terminate in one of many final situations, especially with nondeterministic operators.

The *Do* relation has been intuitively defined in the original specification of Golog[1] as follows: Primitive and Sensing actions:

$$Do(\alpha, s, s') \stackrel{def}{=} Poss(a[s], s) \wedge s' = do(a[s], s)$$

Exogenous actions:

$$Do(\alpha, s, s') \stackrel{def}{=} s' = do(a[s], s)$$

Sequence:

$$Do(\delta_1; \delta_2, s, s'') \stackrel{def}{=} \exists s' \,\big|\, Do(\delta_1, s, s') \wedge Do(\delta_2, s', s'')$$

Test:

$$Do(?(\phi), s, s') \stackrel{def}{=} \phi[s] \wedge s = s'$$

Non-deterministic branch:

$$Do(ndet(\delta_1 \,\big|\, \delta_2), s, s') \stackrel{def}{=} Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$$

Non-deterministic arguments:

$$Do(\pi(\vec{x})\{\delta(\vec{x})\}, s, s') \stackrel{def}{=} \exists \vec{x} \,\big|\, Do(\delta(\vec{x}), s, s')$$

Non-deterministic iteration:

$$\begin{aligned} Do(\delta^*, s, s') \stackrel{def}{=} & \forall P \mid \{ \\ & (\forall s_1 P(s_1, s_1)) \wedge (\forall s_1, s_2, s_3)\, [P(s_1, s_2) \wedge Do(\delta, s_2, s_3) \supset P(s_1, s_3)] \\ & \} \supset P(s, s') \end{aligned}$$

That is, we consider the set of pairs of situations (s, s') where executing δ zero or more times transforms the situation from s to s'. The set is defined inductively:

- Base case: the pair (s, s) must be in the set for all situations s since this indicates zero executions of δ
- Inductive step: if the pair (s_1, s_2) is in the set and performing δ in situation s_2 transforms the situation to s_3 then zero or more executions of δ must be able to transition from s_1 to s_3, so (s_1, s_3) must also be in the set

$P(s, s')$ can then be thought of as indicating that the transition from s to s' exists and can be achieved by some number of executions of δ, which corresponds to an intuitive view of what the *Kleene star* should do.

We can now declare some imperative constructs in these terms:

If statements:

$$Do(if(\phi)\{\delta_1\}else\{\delta_2\}, s, s') \stackrel{def}{=} ndet\,(?(\phi); \delta_1) \,|\, ?(\neg\phi); \delta_2)$$

While loops:

$$Do(while(\phi)\{\delta\}, s, s') \stackrel{def}{=} (?(\phi); \delta)^*; ?(\neg\phi)$$

For loops:

$$\begin{aligned} Do(for(\delta_{\text{init}}; \phi; \delta_{\text{update}})\{\delta\}, s, s') &\stackrel{def}{=} Do(\delta_{\text{init}}; while(\phi)\{\delta; \delta_{\text{update}}\}, s, s') \\ &\stackrel{def}{=} \delta_{\text{init}}; (?(\phi); \delta; \delta_{\text{update}})^*; ?(\neg\phi) \end{aligned}$$

There is also an alternative semantics for the Situation Calculus in terms of *trans* and *final* predicates, which this thesis project can be viewed as extending. This formulation was introduced in a paper by *De Giacomo*, *Lespérance* and *Levesque*[2] which focused on interleaved execution to provide concurrency in the Situation Calculus itself. Due to the interleaving style, the *trans* predicates were constructed such that they mapped a program and a situation (as in the traditional *do* function) to a *subprogram* (the remaining program after executing an initial segment) and a new situation at that point. Thus the language constructs used in a program can be partially executed and not completed until an entirely separate branch of computation is done beforehand. The level of this interleaving is still controlled by the definition of the predicate, such that conditional constructs – particularly *if* statements – execute the condition and the first part of the body atomically. This allows simple implementation of semaphores and other concurrent tools. Alternatively, it also means that thread safety often comes for free and explicit interleaving control is unnecessary.

The *trans* predicates are complemented by *final* predicates which indicate whether a program has successfully completed in a given situation. These two terms can then be combined into a larger system called Do_2. The semantics of this new system have been proved equivalent to those of the original *Do* specification for the Situation Calculus[2]. That is, *Congolog* faithfully implements the Situation Calculus. This proof idea is the basis of our own argument that we have implemented the Situation Calculus; specifically we will prove equivalence with *Congolog*'s *trans* and *final* semantics. The details of this argument will be discussed later.

2.2 Short history of *Golog*

2.2.1 Features

Golog is a planning language with semantics defined by the Situation Calculus. However the details of this implementation are conveniently abstracted away from user code. That is, the syntax of a *Golog* program is entirely independent of situations, without losing the benefits of being based on a formal theory of actions. It also introduces imperative constructs like *if* and *while* as convenient syntactic sugar for the underlying logical representation.

Traditionally, every version of *Golog* has been implemented in *Prolog*, a popular logic-programming language. This is a sensible approach since it offers several powerful features:

- Backtracking search
- Variable unification and pattern matching
- Extensive library support (for personal language extensions)

The original *Golog* interpreter [1] was only twenty lines of *CProlog* code, yet it facilitated a cute (and now canonical) 'elevator' example. However this was largely its limit. Further extensions since then have turned *Golog* into a viable robotics language. A rough summary of this progression is:

- *Congolog*: 'concurrent' (interleaved) execution
- *Indigolog*: external interaction (sensing and exogenous actions), simple planning
- *Readylog*: stochastic actions and decisions[5], decision-theoretic planning

[5]This was retrofitted into *Indigolog* over the 2008-2009 summer, however the implementation begged for a better solution.

2.2.2 Constructs

Indigolog offers several language primitives for developing programs, including some useful though uncommon features such as Kleene star for non-deterministic iteration. The main instructions include:

Construct	*Indigolog* Code	Description
Primitive action α	$\alpha(\vec{x})$	perform an action (if preconditions are satisfied)
Sequence	$[\delta_1, \delta_2]$	perform a sequence of actions or other instructions
Test	$?(\phi)$	succeeds if tested predicate is true
Conditional	$if(\phi, \delta_1, \delta_2)$	traditional if-then-else
Iteration	$while(\phi, \delta)$	traditional while loop
Procedure ρ Call	$\rho(\vec{x})$	traditional procedure call
Procedure ρ Definition	$proc(\rho(\vec{x}), \delta)$	traditional procedure definition
Nondeterministic iteration (Kleene star)	$star(\delta)$	loop an undefined number of times
Nondeterministic choice of action (ndet)	$ndet(\delta_1, \delta_2)$	choose and perform either subprogram δ_1 or δ_2
Nondeterministic choice of argument (π, or pick)	$pi(\nu, \delta)$	choose some binding for variable ν and substitute into δ
Interrupt	$interrupt(\phi, \delta)$	performs program δ if condition ϕ becomes true
Search	$search(\phi)$	determine a sequence of actions to execute to achieve some goal

2.2.3 Limitations

Despite these features, there are several major shortcomings of existing versions:

- **Poor integration:** While *Prolog* allows rapid prototyping, this is a double-edged sword since it means a solution *can* be developed before other features (including syntax or efficiency) need to be considered. The result is an interpreter that works, but is syntactically complex, runs out of stack space and is virtually impossible to debug.

- **Archaic syntax:** Current syntax exploits existing language features (like lists for sequential actions) resulting in verbose and distracting code. *Prolog* offers DCGs that could improve this situation, though equivalent approaches exist in other major languages (such as boost::spirit or yacc).

- **Efficiency:** *Golog* is interpreted by *Prolog*, *Prolog* is (generally) an interpreted language. That is, *Golog* is 'doubly interpreted'. This horrific inefficiency was recognised and led to *CML* [3] – a Cognitive Modeling Language. However this alternative has diverged into mainly animation-centric domains [4]. Alternatively, *Prolog* can be compiled, although this subverts the convenience of an interpreted language without eradicating its overheads.

- **Debugging:** As mentioned above, *Golog* is written in *Prolog* (which has limited debugging support, given its complicated execution) and is doubly interpreted. The result is a language that cannot be debugged without running through *interpreter code as well.* Attempting to *trace* through broken code in this environment quickly reveals itself as a bad idea.

- **Multiple Agents:** *Indigolog* introduced the necessary primitives for the interpreter to interact with the world. However, there is little facility for properly abstracting multiple agents, both in representation and in the communication routes this would necessitate.

It is the balance of these strengths and weaknesses that has inspired this thesis project.

2.3 Cognitive Robotics Languages

The most prominent cognitive robotics tool is STRIPS – the Stanford Research Institute Problem Solver [5] – which remains a major source of inspiration today, despite its age. Originally designed for controlling their robot 'Shakey', it has since been applied to many planning and scheduling tasks. It is based on Propositional Calculus – another logic-based alternative to the Situation Calculus – and features a planner and theorem prover. Modern appropriations tend to focus on only one of these two features, often depending on whether they are targeted at robotic control (or some other dynamic domain). As a final note, STRIPS is technically for problem specification, rather than a control language of itself.

One of the languages to come out of STRIPS is PDDL (Planning Domain Description Language) which endeavours to be a standard encoding for classical planning tasks, or to express the 'physics' of a domain [6]. It was designed as a problem specification language specifically for the AIPS 1998 planning competition; entrants bring their own planners and only have to modify the interface to receive problem specifications in PDDL. In a broader sense it is hoped that PDDL will encourage the 'sharing of problems and algorithms, as well as to allow meaningful comparison of the performance of planners on different problems'[6]. Interestingly, the main omission of this language is 'advice' to the underlying planner – notes about preconditions or suitable actions in a given context – which are generally necessary for effective reasoning. This 'perverse neutrality' [6] is intended, since every planner now needs to extend the language to cope, but is open to do it in whatever way they prefer.

PDDL also comes in several layers, the first being STRIPS planning itself. Successive layers add additional language features including ADL[6]. It also sports universal quantification (though this is syntactic sugar for what *Golog* also supports) and nicely distinguishes domains and problems by certain keywords and by convention of storing them in separate files[7]. Further layers add new functionality including functions, types and object-oriented-like type hierarchies. Many of these features are missing in *Golog* and should be considered by this thesis.

Funge's CML (as previously mentioned) is another alternative. This is significant since it targets the main inefficiency displayed by many languages of this style – an interpreted interpreter – by compiling code into pure *Prolog*. Despite this, it appears to have received little attention and the code itself is no longer available from Funge's website (though a copy can be obtained via the

[6]Note that effect axioms may have conditions (as in *Golog*) or be universally quantified. However, *PDDL*'s ADL component is claimed to be no more expressive than *Golog* [7].

[7]Again, *Golog* is open to this convention since it simply requires importing a separate file. However we found with the iCinema that the domain changed for different problems so the benefit was generally not pronounced enough to warrant it.

Wayback Machine as linked in the References section).

There are also several languages based on situation calculus-like theories such as *Flux* [8] (fluent calculus) which was used to develop *fluxplayer* – the 2006 AAAI general game playing champion. This significant success indicates the potential of these logic-inspired languages and hence of *Golog* too.

Similarly, 'action languages' including the *Causal Calculator* [9] (non-monotonic causal logic) exist, which focus on problems of 'action and change'. These problems are often transition systems that can be modelled as automata-like graphs. The *Causal Calculator* appears otherwise very similar to *Golog* in terms of syntax and (*Prolog*-based) implementation.

An alternative system for robotic control is in the form of Reactive Action Packages (RAPs) [10] which recognise that planning a sequence of robot actions is generally not sufficient – many situations may occur that irrepairably interrupt the sequence. RAPs serve as 'hierarchical building blocks' for creating plans that select an appropriate action at each point in time. This is done in the presence of an execution monitor, thus avoiding the need to replan on failure. Interestingly, there is a parallel between this and *Golog*'s nondeterministic features: a *Golog* program can request the interpreter to do one of two (possibly nested) streams of execution[8]. This is achieved by attempting the first stream; if this fails then it tries the second. The result is that the interpreter 'finds' an action that works, even if the world changes between calling and executing.

Nilsson (who was involved in STRIPS) has also produced an interpreter for Teleo-Reactive (T-R) Programs that execute by dynamically generating 'circuitry for the continuous computation of the parameters and conditions on which agent action is based' [11] to introduce 'circuit semantics' to program execution. The rationale is that control theory and circuits themselves are better at dealing with continuous change than traditional computer science constructs like functions and sequences. Yet continuous change is precisely what robots in dynamic domains must endure.

Finally, there are a number of programs based on the Beliefs-Desires-Intentions (BDI) model of AI of which *AgentSpeak* [12] is a popular abstract formalisation. Specifically, *AgentSpeak* aims to produce a formal model for dealing with BDI and to reconcile this with a desire for concurrent agents working together to solve problems.

AgentSpeak has since inspired *AgentSpeak(L)* – a logic-based version similar to *Congolog*, although not as 'rich' [13]. This has in turn led to open-source versions such as *Jason* which has extended the language to include support for distributed agents, plan failure-tolerance and strong-negation[9] [14].

An alternative BDI implementation is the Procedural Reasoning System (PRS) [15] which

[8]This is a helpful feature inherited from *Prolog*.

[9]This enables it to handle both open-world and closed-world assumptions.

integrates reactivity into traditional planning methods. This allows *PRS*-based agents to 'survive in highly dynamic and uncertain worlds'.

Jack is another BDI system. It features 'teams' for modelling social structures and coordinating group behaviours as well as inter-agent coordination and resilience against plan failure. Incidentally it is produced by the AOS Group, who have applied *Jack* to several international military products including tactical analysis, training and autonomous control of vehicles[10] [16].

[10] *Jack* successfully provided 'autonomous vehicle management' for a BAC1-11 airliner in 2007. AOS Group are now committed to an unmanned stealth combat aircraft – the 'Taranis' – by 2010[16].

2.4 Games

Although commercial games provide little disclosure of their techniques there are some internet communities that give some insight into current industry methods: for example, AIGameDev.com [17]. Currently there tends to be a focus on scripting engines – generally Lua or Python based – even in large-scale games like *World of Warcraft* and *Crysis*. This is probably due to the programming convenience it provides for tweaking parameters and controlling predetermined storylines. Performance gains are generally made by delegating intensive routines to specific C++ implementations, as in *Crysis*.

Besides this, most game AI reduces to searching problems, so A*, minimax/negamax and $\alpha\beta$-pruning all claim popular usage.

Other standard artificial intelligence constructs like behaviour trees and finite state machines also make regular appearances.

There is one notable exception where a successful, commercial game took inspiration from a formal cognitive language: *F.E.A.R* (a fast-paced first-person shooter with very competitive agents) is alleged to be largely based on STRIPS. However, the particular implementation is said to have been heavily optimised to make planning worthwhile compared with a purely reactive (but very *responsive*) agent. It is likely that this technology is uncommon due to the difficulty of achieving target performance, along with a lower standard for agent intelligence. Despite this, *F.E.A.R* presents a case where *Golog*-like languages can produce superior results, regardless of their arguably more complicated representation or speed limitations.

Additionally, *Readylog* has been successfully interfaced with *Unreal Tournament* – another popular (though older) first-person shooter – to provide competitive players [18]. As with *F.E.A.R*, this implementation was only effective after heavy optimisation, this time in the form of 'precompiled macros' to speed up sections where repeated planning would throttle performance.

2.5 Agent Communication Frameworks

2.5.1 FIPA

FIPA – the Foundation for Intelligent Physical Agents – is the eleventh IEEE standards committee, comprised of over 50 affiliated institutions including universities, telecommunication companies and other prominent organisations. It endeavours to produce agent-oriented standards, with a particular focus on agent communication [19]. The standards it has produced detail a convenient, extensible format for message creation, passing and response. They also describe the concept of a 'yellow pages' for service discovery, where an agent can register its services on a publicly available database. Other agents that require this service can then query the database to discover the service provider and how to contact them.

2.5.2 OAA

The open-source alternative to FIPA is the Open Agent Architecture, which similarly defines a flexible message format and a means of agent discovery. A major distinction is that OAA describes a 'blackboard' server holding lists of tasks. Agents then subscribe to this blackboard and perform listed tasks if they are able to.

Chapter 3

Target Platform

3.1 Java

Java is a popular high level programming language. Java programs are written in a clear, C-like syntax and compiled to Java bytecode – architecture independent files that are executed by a Java Virtual Machine (JVM). This format allows for the Java 'compile once, run anywhere' paradigm, making Java one of the most useful, portable and pervasive modern languages.

Internally, Java performs stack-based execution of bytecode programs, that is, operators like *iadd* (add two integers) or *invokevirtual* (function call) push and pop data to and from a runtime stack to achieve program execution. For example, to perform $3 + 5$ we push the integer constants 3 and 5 onto the stack then perform the addition instruction, which pops the top two operands (3 and 5) performs the addition, and pushes the result, 8, back onto the stack:

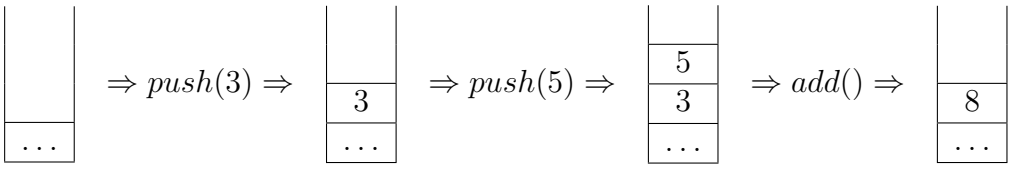

Table 3.1: Java stack transition for simple addition

Java functions are performed in the same way: an object reference (for non-static functions) and any argument values should be pushed to the stack (in that order) and then the function call

instruction is executed. For example, consider that we have a reference to a person object *this* which has a method in Java syntax like:

$$String\ \mathbf{name}(int\ i);$$

The corresponding stack activity would then be:

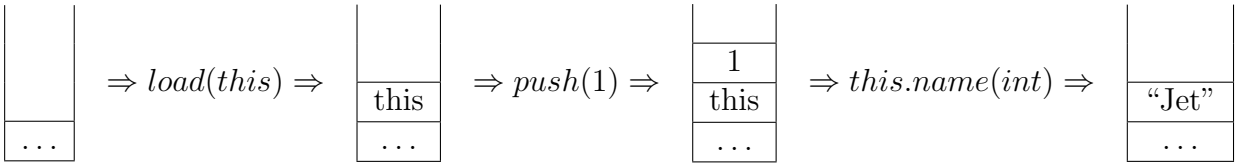

Table 3.2: Java stack transition for function call

Note that the final stack snapshot contains the string "Jet"; this is the function return value.

In addition to the stack for operands we also have a direct-access memory system. Local variables can be stored to and loaded from numbered 'variable slots'. For example, we can load the number 3.14 and store it into local variable 2:

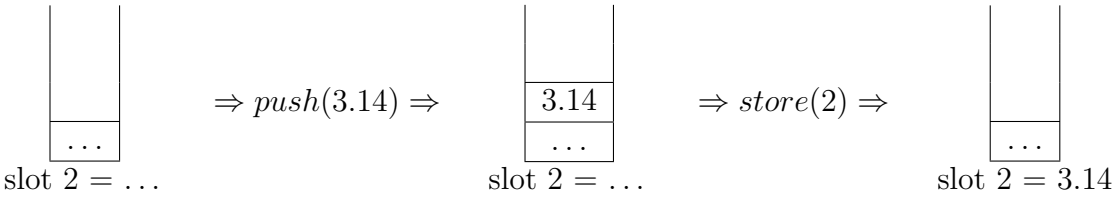

Table 3.3: Java stack transition for storing a variable

We may then perform some arbitrary computation with the stack. After this we may need to retrieve the value we stored, so we can load its value back onto the operand stack:

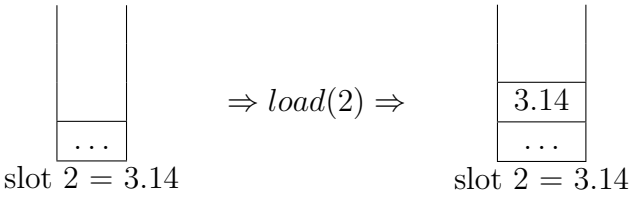

Table 3.4: Java stack transition for loading a variable

We can see that a variable slot's value is undefined until it is assigned although most JVMs will complain if you read it uninitialised. Also, the assigned value remains in the slot until it is overwritten. Note that each function gets its own variable memory, so loading or storing with a slot in one function will have no effect on any other functions.

Finally, the process of calling a *Java* function defines the first few memory slots within that new function. For example to call *Object*'s .equals method we do the following:

Listing 3.1: invoking a virtual function in *Jasmin*

```
1    push Object
2    push Argument
3    invokevirtual java/lang/Object/equals(Ljava/lang/Object;)Z
```

where *java/lang/Object/equals* indicates the 'equals' method of the *Object* class, *Ljava/lang/Object;* is the *typename* of the argument and *Z* is the return type (Boolean). Note that we push on the *Object* first, followed by the function arguments (*Argument*). These will be stored in local variable slots zero and one respectively of the new function. Additional local variables will be stored in subsequent slots – part of the same memory system. This highlights the *Java* 'call convention' which says that virtual (non-static) functions need to receive the 'this' reference first. We have already seen this in the first example: loading the 'this' reference is actually achieved with the *Java* mnemonic/*Jasmin* instruction:

aload_0

where the *a* in *aload* indicates a reference type and 0 indicates the variable slot. Of course, invoking a static function does not require providing a 'this' reference. Function arguments (or locals if the function does not accept arguments) will thus start at zero. Given this, it is best to consider all arguments (including the 'this' reference) and local variables in the same way, since no distinction is given in *Java* bytecode, *Jolog* or normal *Java* code itself.

3.2 Java Exceptions

Java exceptions are declared in a familiar *try-catch* statement. However the virtual machine needs to be able to transfer execution from an arbitrary point in a try statement to its exception handler (the catch clause). To facilitate this, the bytecode translates these structures into entries in an 'exception table' for each function definition:

type	from	to	with
$java/lang/NullPointerException$	myTryBegin	myTryEnd	myCatch
$jolog/FailedPreconditionException$	beginNdet	endNdet	failNdet
all	here	there	anywhere
...			

When the virtual machine executes a *throw exception* instruction, it looks up the first entry in the exception table at the current scope with the corresponding type and instruction range. If a matching entry is found then execution is transferred to the instruction specified in that entry, otherwise the virtual machine breaks to the previous scope and tries again. If it repeats this process until it runs out of functions then this corresponds to the entire program failing with an uncaught exception.

Let us now consider the above exception table. If a NullPointerException is thrown between the labels $myTryBegin$ and $myTryEnd$ in user code then the JVM will look up this table and see that the first entry works. It can then transfer control to the first instruction after the label $myCatch$. Note that the 'type' field of the exception table must contain a valid *Java* class. The one exception to this rule is the keyword *all*, which instructs the JVM to ignore the exception type when determining whether the corresponding entry matches the thrown exception (that is, it is sufficient for the exception range to match; any arbitrary exception type will be accepted by the handler). We can observe now that the third exception handler may also match our NullPointerException, however, the JVM accepts the earliest matching entry, so the specific NullPointerException handler will be used in preference to the *all* handler by virtue of its position in the table. This is actually just a special case of the more general inheritance mechanism that these exception handlers support; the first entry that matches may have a different type, but it can still match if it is a superclass of the actual exception type. These two phenomena are familiar in normal *Java* code too: catching a $java/lang/Exception$ before a $java/io/IOException$ 'shadows' the $IOException$ – it will never get called since the earlier more general exception is able to match first.

Java's internal representation of exception handling code means that *Jasmin* assembly does not include *try-catch* statements. Instead it requires that you explicitly state each exception entry manually, in the form[1]:

$$.catch\ Type\ from\ start\_label\ to\ end\_label\ with\ handler\_label$$

[1]Note that *start_label* and *end_label* are labelled points in the assembly code defining a region of code where this exception may occur and *handler_label* is the address (label) to transfer execution to when 'catching' the exception.

This structure allows a simple definition of a hierarchy of code regions, each of which performs some task and returns or passes execution further down the exception table on failure. This was essential for developing our model of nondeterminism and will be reintroduced later when we discuss the details of this implementation.

3.3 *Jasmin*

Jasmin is a popular, open-source *Java* 'assembly language' with a convenient textual formulation that closely mimics the *Java* classfile bytecode mnemonics. This makes it an ideal intermediate language between *Jolog* and *Java* bytecode. Accordingly the *Jolog* compiler actually produces *Jasmin* assembly files. The final task of converting this assembly language into runnable *Java* classfiles is delegated to *Jasmin*.

Some number formats and character escape sequences are unimplemented in *Jasmin*. The newer version *Jasmin XT* is much more comprehensive and behaves correctly with these inputs.

3.4 Datatypes

Type	Range	Syntax
Void		V
Boolean	{0, 1}	Z
Byte	8-bit signed integer	B
Short	16-bit signed integer	S
Integer	32-bit signed integer	I
Long	64-bit signed integer	L
Char	16-bit unsigned Unicode	C
Float	32-bit IEEE 754 single precision	F
Double	64-bit IEEE 754 double precision	D
	References	
Return address	32-bit unsigned reference	not permitted
Class Reference		Lpackage/classname;
Interface Reference		Lpackage/interfacename;
Array Reference		[type

Note that references are to a type – possibly a reference type. For example a 3-Dimensional array of Strings would appear as `[[[Ljava/lang/String;`

Methods also have a strict format:

`package/name/classname/method(arguments)return`

Note that the argument types are placed consecutively, without any delimiters.

Examples:

Java:	package java.io; class PrintStream { void println(String); }
Jasmin:	`java/io/PrintStream/println(Ljava/lang/String;)V`
Java:	package my.package; class myClass { int foo(Object, int[], double); }
Jasmin:	`my/package/MyClass/foo(Ljava/lang/Object;[ID)I`
Java:	package package; class myOtherClass { int foo(Double, double); }
Jasmin:	`package/MyOtherClass/foo(Ljava/lang/Double;D)I`

3.5 *Jasmin XT* Enhancements

The first version of *Jasmin* has since been extended with certain new features:

- absolute and relative offsets can be used in addition to labels as branch targets, local variable visibility ranges and exception handlers
- `.bytecode major.minor` directive to set bytecode version in classfiles
- `.inner` directive for inner classes?

None of these are utilised by *Jolog*. However this new version is valuable for the improved stability, particularly with encoding Unicode escape sequences.

Chapter 4

Jolog

Jolog is a cognitive, robotics language. Specifically it is a new object-oriented, Java-like formulation of Golog. This is complemented by the Jolog compiler capable of compiling Jolog programs into Java bytecode. These programs can then be run on any Java Virtual Machine (JVM) on any platform.

This has many favourable implications:

- *Jolog* is extremely portable
- More familiar syntax[1] accelerates development and removes many of the syntactic pitfalls of previous Gologs[2]
- Easier debugging – most Golog-interpreter duties have been delegated to the Java Virtual Machine, significantly reducing interpreter code to trace through. Further, Jolog programs compile to pure Java bytecode, so they are compatible with existing-Java debugging tools.

Jolog is written in C++ using the Boost::Spirit parser library. Extensive use of meta-templates allows a grammar resembling Extended Backus-Naur Form (EBNF) to be written directly in native C++ code. Input files can then be directly parsed into tokens – atomic words, identifiers, literals, etc. These tokens are then converted into Abstract Syntax Trees (ASTs) – unambiguous groupings of tokens into more useful structures. The important information is then extracted from these trees

[1] We suggest that Java-like syntax is familiar to a broader audience than Prolog-like syntax.

[2] Particularly in regard to the declaration and use of Prolog variables and other ill-defined Golog-Prolog boundaries. My favourite problem arises because the Golog interpreter does not provide a variable scheme that is sufficient in all cases, so the use of Prolog variables is essential for some complicated tasks. However because the interpreter controls execution it may occasionally rerun a section of code. The problem then is that a Prolog variable may be successfully bound on the first run, but then the *same*, bound variable will be reused in the next execution. The rerun code is unable to complete now because it has no free variables, causing the program to silently fail. This took a long time to discover.

and reassembled into a secondary 'meta' tree of 'constructs': *if* and *not* nodes for example. These constructs know what their final assembly code should look like and are responsible for 'emitting' their precise representation in Jasmin assembly code to the output file. Jasmin is a popular open-source compiler from a simple assembly language to Java bytecode. Hence, translating from Jolog to Jasmin and then to Java bytecode alleviates the need to produce binary Java class files (Jasmin does this for us). Further, Jasmin is fairly pervasive in the Java compiler community, crediting itself as 'the de-facto standard Java assembly language'[20].

4.1 Constructs

We have previously seen how *Indigolog* constructs are written. We now briefly introduce the *Jolog* versions. A longer discussion of their syntax and implementation is left for later chapters.

Construct	*Jolog* Code
Primitive action α	$\alpha(\vec{x})$;
Sequence	$\{\ \delta_1\ \delta_2\ \}$
Test	$?(\phi)$;
Conditional	$if(\phi)\ \{\ \delta_1\ \}[else\ \{\ \delta_2\ \}]$
Iteration	$while(\phi)\ \{\delta\}$
Procedure ρ Call	$\rho(\vec{x})$
Procedure ρ Definition	$type\ \rho(\vec{x})\ \{\ \delta\ \}$
Nondeterministic iteration (Kleene star)	δ^*;
Nondeterministic choice of action (ndet)	$ndet(\delta_1 \mid \delta_2)$
Nondeterministic choice of argument (π, or pick)	$pick(\nu)\ \{\delta\}$

Internally *Jolog* actually considers the *Sequence* as a *Scope*. Furthermore it allows an arbitrary number of component instructions, however, we present only the two element sequence here for reasons of space and to ease of our proof later on. Note that *Indigolog* is guilty of the same simplification although we suggest that it does not affect the expressiveness of the construct, neither here or in *Indigolog*.

4.2 Language Features

Jolog exhibits all of the core features of existing Golog languages including backtracking, pattern matching for exogenous actions and non-deterministic execution. *Jolog* also introduces several new features:

- Agent communication primitives
- Easier agent abstraction
- More familiar threading model[3]
- Ability to directly call native Java code, including any Java library
- Object-Orientation

There are some differences though. Function pattern matching – a distinctive feature of *Prolog* – was inherited not so much because it was appropriate but because it was the easiest way to implement procedure calls. A *Prolog*-like variable class has been implemented that could be used for this purpose, however it is not currently activated and has to be completely monitored by the user – including explicitly unbinding before reuse.

A similar omission is with backtracking functions. *Prolog* allows control to backtrack into functions that have *already completed*. This feature has been reproduced in *Jolog* with some effort. However the ability to define several functions and the compiler backtrack through each one has not. This design decision relates to the idea of multiple functions – in *Prolog* each of these 'procedures' is actually a rule, so multiple definitions make sense. Further, eligibility of a match is done on the name of a rule and its arity. This becomes much harder (or at least more restrictive) when types are made explicit as in *Jolog*. However while the compiler will not do this for free anymore, it can still be emulated via the *ndet* operator:

Listing 4.1: emulating *Prolog*-style alternative functions in *Jolog*

```
1 ndet( function1(argument); | function2(argument); )
```

[3] *Java* thread constructs replace those available in *Congolog*. One could argue this is a step backwards in terms of expressiveness – prioritised concurrency is probably simpler to manage in *Congolog* than *Java*. However we are less concerned with concurrency within a single program or agent. Our focus is on more 'industrial' *Java* threads (which are provided by the operating system in modern JVMs) capable of running many single-threaded agents simultaneously. This model is certainly more appropriate for the *iCinema*.

4.3 Object-Orientation

We used *Indigolog* over the 2008-2009 summer to develop agents for the *UNSW iCinema*. Sadly these multiple-agent systems were developed within a single *Indigolog* program, partially to ease distribution to the *iCinema*. More specifically, one execution had to control several agents at once. While this was certainly possible, it also made the code murkier and introduced many errors. The communication between agents in this setting was also better achieved by 'cheating' – agents could inspect global state instead of asking other agents and storing an internal model of the world. This thesis has emerged from a desire to isolate these individual agents into separate objects with their own goals and knowledge about the world.

The reasons for producing an object-orientated Golog stem from the above scenario and can be summarised as:

- Object-orientation provides a more intuitive mechanism for modeling our target domains.
- We get cleaner agent abstraction and better opportunity for agents to communicate properly.
- It is (in some senses) more efficient, since we get better code reuse, functionality inheritance and smaller 'knowledge bases'[4].

Finally, we recognise that *Java* is an ideal platform for introducing this object-orientation:

- The JVM is designed to handle many objects with many classes.
- *Java* is a widely-known and widely-supported language, with an impressive back catalogue of supporting libraries and extensions.
- Distribution of multiple agents and support code is simple via compressed *.jar* files – they do not even need to be unpacked or recompiled.
- *Java* runtimes are available on almost every system, even amongst non-development home computers.

[4]The term 'knowledge base' relates specifically to *Prolog*. In our case we would have smaller 'fluent stores' (which will be introduced later). The basic intention is to have more agents with less fluents; less fluents means less values – and less *spurious* values in particular – to search through and hence faster search times.

Chapter 5

How *Jolog* works

Jolog seeks to delegate as much execution to the virtual machine as possible; a reaction to debugging code in interpreted Golog languages. This means pure Java bytecode and no execution monitor. Jolog code is hence compiled fairly directly: Jolog functions appear as Java functions might. The distinction is in the additional features that Jolog provides, particularly fluents and actions. These are given very lightweight infrastructure such that they do not introduce a significant increase in bytecode size but still permit the behaviour expected from previous Gologs. The implementation of these features is the subject of this chapter.

Golog consists of:

- Initial situation
- Action Theory[1], Σ
- Program
- Exogenous actions (since Indigolog)
- Sensing actions (since Indigolog)

The liveness of Golog programs hinges on 'trans' statements – transitions from a sequence of interpreted predicates to a new sequence representing the next state. The advantage of compiling to Java bytecode is that this execution can now be handled entirely by the JVM.

Similarly, Golog programs determine whether they have completed – successfully or otherwise – via 'final' statements, which update the notion of program completion just as the precise program

[1]The action theory incorporates several rules and axioms – including the unique name assumption – to overcome many restrictions and inconveniences that first-order logic otherwise implies. It also more obviously includes a specification of actions, each with preconditions, effect axioms and more.

state is updated by trans statements. Java has two forms of completion: successful termination after running all code and returning from the first function (usually *main* or *run*), or failure if an unhandled exception – thrown by a failed action for example – forces premature completion. Exceptions form an elegant method of replacing final statements and introducing backtracking function calls to Java: invalid functions will fail, throw an exception back to the point at which they were called and a successor function can be tried. If a function can be called at this choice point then eventually it will be and execution will continue towards successful completion. If no valid execution exists then the last exception will be uncaught and we will regress to an earlier choice point. If no early handler exists then the program has no valid execution sequence – it has failed – and so it will terminate, possibly with an unhandled exception from the first function.

Backtracking with exceptions is not as simple as this. In particular it requires a method of saving and restoring 'choice points'. The details of this implementation are discussed later in this chapter as part of the section on nondeterminism.

Finally, Golog maintains a current data state as well (the situation): a collection of received and performed actions that can be combined with the action theory to determine the value of any fluent. In Indigolog this was achieved by simulating the effect of every received action in the history on the initial value. This implementation was clearly a severe inefficiency, especially in the presence of many actions and was exemplified by the fact that we committed to an action immediately, so we might as well have updated the fluent directly. This was improved by 'rolling' the database forward (replacing the initial condition for a fluent with its value at a point in the history and then clearing the history up to that point) but it remains less effective than a simple lookup of a local variable. In Jolog we take a compromise: all fluents are stored in the FluentStore – an object with an internal HashMap for fast lookup of fluents. Fluent updates are commited to immediately and are stored and retrieved via short function calls. The FluentStore is also written in higher-level Java code (as opposed to hand-written assembly) so it can be better used for harder tasks (pick, search). However it also represents our situation (the collection of fluent values), an integral component to the implementation of the Situation Calculus.

5.1 Overall System

The Jolog compiler undergoes several distinct phases to produce the compiled Jolog file. The brief overview appears as:

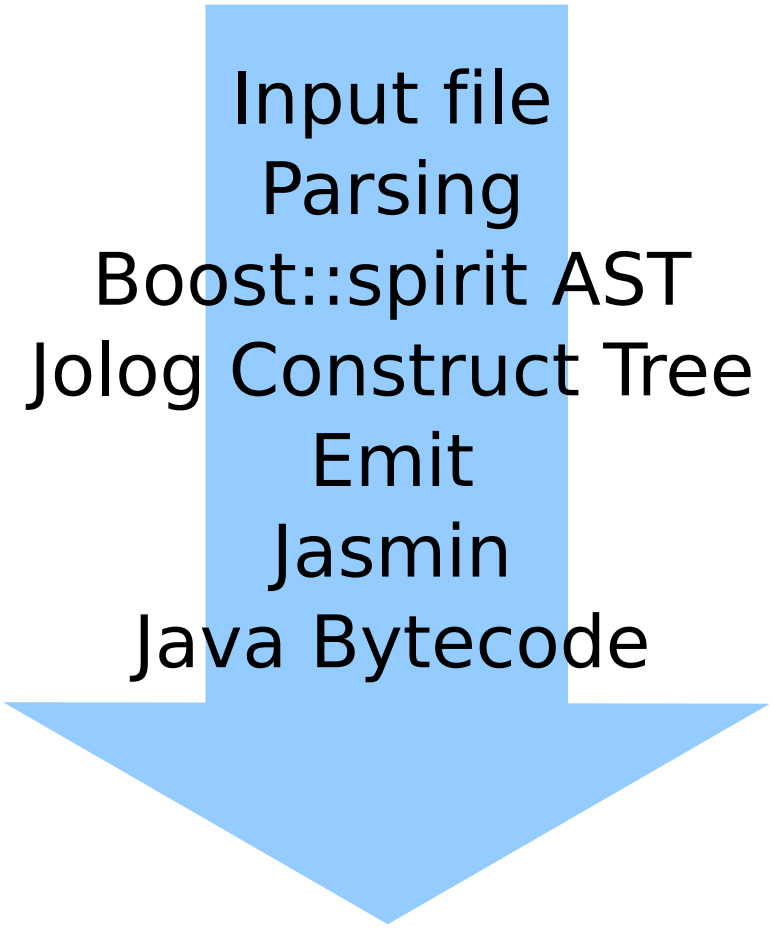

Figure 5.1: Phases of compilation

In more detail: a source file is opened and given to the *boost::spirit parse_ast* function, which parses this file and creates an abstract syntax tree (AST) from the resulting tokens. That is, a tree containing substrings from the initial file in a hierarchy that groups grammatically related tokens together. We then disassemble this tree and use the relevant pieces to create a tree of language constructs, a 'Jolog Construct Tree' (JCT). That is, if the AST holds something like the following:

```
ID: if statement, Value: <if>, Children (3)
   ID: postfix, Value: <.>, Children (2)
      ID: identifier, Value: <s>, Children (0)
```

```
      ID: postfix, Value: <(>, Children (2)
         ID: identifier, Value: <equals>, Children (0)
         ID: cast, Value: <(Object) "password">, Children (2)
            ID: type, Value: <Object>, Children (0)
            ID: string literal, Value: <password>, Children (0)
   ID: scope, Value: <{>, Children (6)
      ID: local variable, Value: <String exclamations;>, Children (2)
         ID: type, Value: <String>, Children (0)
         ID: identifier, Value: <exclamations>, Children (0)
      ID: local variable, Value: <//do it with a for l ...>, Children (2)
         ID: type, Value: <int>, Children (0)
         ID: identifier, Value: <i>, Children (0)
      ID: for statement, Value: <for(i=0; i<2; i=i+1) ...>, Children (7)
...
```

Then this sub-AST will produce a corresponding *if* node in the JCT:

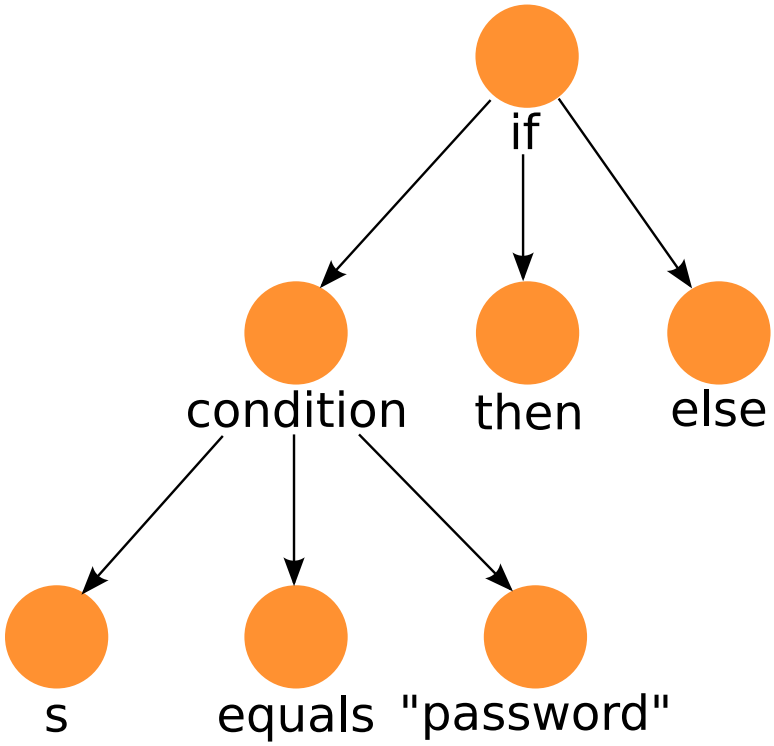

Figure 5.2: Example *Jolog* construct subtree

This *if* node would form part of a larger tree of constructs – hence a construct tree – with an *Agent* node as the root. Once this tree is finalised we can 'emit' the tree – write its corresponding assembly to a file. The advantage of this method is that each individual construct need only know a small amount of assembly to achieve what it needs to, yet the overall system is still capable of complex computation. In this case, the root *Agent* would emit necessary class initialisation, the constructor, then delegate definitions of each of its functions to child *Function* nodes. These will in turn emit a function prototype and then delegate the specifics of its body to the individual constructs it contains, such as *scope* nodes[2], *if* nodes, *function call* nodes, etc.

The final stage of compilation is achieved by the external *Jasmin* compiler. This is a third-party Java program that accepts a file – assembly code produced by the *Jolog* compiler – and produces a corresponding Java classfile.

5.2 Boost::Spirit

Spirit is a parser framework distributed as part of the boost family of C++ libraries. It exploits C++ metatemplates and operator overloading to allow users to write grammars in a format closely resembling extended Backus-Naur form. The result is C++ code that looks like the following:

Listing 5.1: excerpt from *JologGrammar.h*

```
1  using namespace boost::spirit::classic;
2
3  class JologGrammar : public grammar<JologGrammar> {
4     public:
5        template <typename ScannerT>
6        class definition {
7           public:
8              definition(JologGrammar const& self) {
9                 jolog_file
10                   = !jolog_package
11                   >> *jolog_import
12                   >> !jolog_agent
13                   >> (end_p | (lexeme_d[ +anychar_p ][self.report(file_tag)] >>
14                   ;
15
```

[2] *Scope* in *Jolog* corresponds to 'sequences' in previous Gologs.

```
16                  jolog_agent
17                     = *prefix >> str_p("agent")
18                     >> (typeName | error[self.report(type_tag)])
19                     >> ( *( implements | extends )
20                          >> no_node_d[ ch_p('{') ]
21                          >> *(jolog_agent_contents)
22                          >> no_node_d[ ch_p('}') ])
23                        | error[self.report(agent_tag)]
24                     ;
25                  //additional rules removed for brevity
26               }
27               rule<scanner_t, parser_tag<file_tag> > const& start() const {
28                  return jolog_file;
29               }
30
31            private:
32               IdentifierGrammar identifier;
33
34               rule<ScannerT, parser_tag<file_tag> > jolog_file;
35               rule<ScannerT, parser_tag<agent_tag> > jolog_agent;
36         };
37   };
```

JologGrammar.h defines our top-level rule (*jolog_file*), so it also provides an interface function, *parseString*, for use by the compiler frontend. This function takes an input string (from a *Jolog* source file) and the name of the file. This information is then converted into an iterator range and passed to a boost::spirit utility function *ast_parse* which parses the input file and produces an abstract syntax tree. Note that we use a special iterator type 'position_iterator' so that nodes in the AST contain position information (filename and line number) which is essential for producing useful error messages.

The *Boost::spirit* parsing routine can be invoked with code equivalent to:

Listing 5.2: excerpt from *JologGrammar.h*

```
1  typedef position_iterator<const char*> iterator_t;
2  typedef node_iter_data_factory<int> factory_t;
3
```

```
4  tree_parse_info<iterator_t, factory_t> parseString(const char* s,
5                                                     string filename) {
6      iterator_t begin(s, s+strlen(s), filename);
7      iterator_t end;
8      SkipGrammar skip;
9      return ast_parse<factory_t>(begin, end, *this, skip);
10 }
```

5.3 Type and Error Checking

The lack of type or error checking in *Indigolog* allowed many trivial errors to manifest into large delays in development. The desire to address this problem helped inspire this thesis. The complexity of this task means that it is spread across several phases of the compilation process. Firstly, error checks need to clearly indicate simple syntactic errors during parsing so that a user can produce a file that matches the grammar. Once the input file is of this standard it can be parsed into an AST and more sophisticated reasoning can be achieved, particularly with respect to type and logic errors.

Boost::spirit allows easy definition of grammars in a fairly intuitive format, but this assumes a correct input. Unfortunately even a minor syntactic error may simply not match the grammar and so *boost::spirit* will get upset and fail with no indication of what is wrong. Worse is that little facility is provided to report on problems as they occur. This was somewhat of a problem in *Indigolog* and was a motivating force for this thesis, so a solution of some kind was essential. The adopted approach is to use *spirit*'s 'semantic actions' – which allow data to be collected and sent to an error-checking function *during* parsing – and empty parsers to catch boundary cases. For example, a missing semicolon after an expression statement triggers the last branch of the statement rule:

Listing 5.3: statement rule from *StatementGrammar.h*

```
1 statement
2     = jump_statement
3     | local_variable_definition
4     | if_statement
5     ...
6     | scope
7     | !expression >> (no_node_d[ ch_p(';') ])
```

```
8       | ((+(alnum_p - '}'))[self.report(expression_statement_tag)])
9       ;
```

A missing semicolon will result in none of the subrules matching, yet the statement grammar is used where a statement is required (and no alternative will suffice). That is, the input does not match the grammar. To combat this, we introduce a final 'error' rule that *will* match when all else fails. This rule takes any trailing alphanumeric characters (via the *alnum_p* parser) and reports them to the user. This reporting procedure is complicated by self-imposed requirements that it write to a given output stream and that it modifies a flag to indicate failure during parsing. Thus a special functor is created by the *report(expression_statement_tag)* function. The collected characters are then given to this functor and complemented with the tag type (in this case *expression_statement_tag* which is used to indicate a missing semicolon to the error system) and the current line number to produce a useful error message.

Inputs successfully matching the grammar are then translated into the 'Jolog Construct Tree' as mentioned above. This provides another opportunity to detect illegal input. For example, when the compiler receives an 'if' node in the AST it will begin constructing an *if* node in the JCT. This involves looking at the children of the AST node. The first child is compiled into a Boolean expression and is used as the *if* condition. If there are no children then the compiler prints an error to the user complaining that the definition is incomplete. Similarly if there are too many children then the compiler complains that it does not know what to do with the trailing code.

Finally, error checking is also performed during the code generation phase. The *if* construct, for example, stores up to three children: the 'condition', a 'then' body and an 'else' body. The type of the condition node is then queried before emitting to ensure it will give a Boolean result. This type of check is essential if we want useful feedback since the Java Virtual Machine will detect the problem if we do not. However the JVM is also far more cryptic in its error messages. Note also that these checks must wait until this late stage because many constructs (such as function calls) won't know their type until everything is defined. The best example of this is a function call to a user function. The function has to be created and register itself with the current *NamespaceResolver* before the function call can determine what type it will get back.

5.4 Fluents

Fluents are achieved via a 'fluent store' – a dedicated object that holds all fluents and provides an interface for loading and storing fluents. Each Jolog agent has a private class variable called *jolog_fluent_store* which holds its own private fluent store. On startup each agent creates and initialises a new fluent store with the initial fluents specified by the initialisation axioms in the Jolog file. This fluent store is then saved into the dedicated class variable.

As mentioned earlier, fluents are accessible anywhere within the parent agent. However they may be modified only by actions. This is enforced by the *Jolog* compiler. The update process is quite simple though – calling an action α executes code equivalent to:

Listing 5.4: *Jolog* successor state axiom pseudocode for action α

```
1    for each fluent f:
2        if action α modifies f:
3            if precondition holds:
4                effect axioms are evaluated
5                assign/overwrite value of f
```

where *precondition* is an arbitrarily complex Boolean expression.

We now consider the value of a fluent in a given situation. First note the role of the situation is to define the value of fluents – it has no other function – and modern Gologs actively enforce this restriction. This means that you cannot query the contents of the situation, just the *current* value of a fluent. This story is only marginally complicated by allowing execution to backtrack. Under these circumstances, we essentially remove the most recent action from the situation.

Updating the fluent store can be observed separately to the semantics of the language itself. Accordingly we present an intuitive argument of equivalence with the Situation Calculus, starting with the case without backtracking, then presenting an argument for extending this implementation to handle situation regression.

As a final note, we observe that the fluent store is implemented as a map of *Java Object*s – reference types unable to represent primitive types like *int*s. To combat this restriction, *Jolog* performs implicit behind-the-scenes conversion from any primitive type to its corresponding reference type and back when loading and storing fluents of primitive type. We will henceforth assume that internal type errors will not occur, particularly those associated with an artficial conflict between primitive and reference types.

5.4.1 Without Backtracking

We start with an assumption about target applications:

Assumption 1: The target applications of *Jolog* are well-defined virtual worlds[3].

This means that the closed world assumption is a convenient logical presence since it allows a simple (and in our case intuitive) solution to the frame problem – the world is already fully defined, so the absence of an effect axiom for a particular fluent-action pair simply means that action does not affect that fluent.

Assumption 2: We are informed of exogenous actions having taken place; they have *already changed the world*, so these actions cannot be backtracked[4].

We first consider the normal 'no backtracking' case and later discuss some details of a hypothetical implementation where actions are allowed to backtrack. To this effect we examine our own implementation, where the fluent store overwrites fluent values so it only stores fluent bindings for the *current* situation.

Assumption 3: We now accept *Reiter's* specification of a situation. Specifically, it is a *sequence* of actions such that if $do(\alpha, s) = do(\alpha', s')$ then $\alpha = \alpha'$ and $s = s'$. To this end we consider actions as name-timestamp pairs.

Assumption 4: The action theory Σ contains successor state axioms (for each action α, fluent f and value v) of the form:

$$\Big[f(do(\alpha, s)) = v\Big] \equiv \Big[\Big(P_f(\alpha, s) \wedge effect(\alpha, s) = v\Big) \vee \Big(f(s) = v \wedge \neg N_f(\alpha, s)\Big)\Big]$$

We are aiming to eliminate the situation term, so we will use Assumption 3 to convert situations – sequences of states – into sets ordered by timestamp. Note that by Assumption 4 these elements should be unique. Now, for $i \in [0, n]$ if we have actions α_i (where i indicates *relative* timestamp

[3]Specifically the UNSW *iCinema* or any other virtual arena with fully-defined world rules, like a computer game

[4]In fact, this holds for primitive actions too – they may modify fluents which are later sensed externally. *Indigolog* and hence *Jolog* both prohibit backtracking of any action for this reason.

ordering) then:

$$do(\alpha_i, s_i) = s_{i+1}$$
$$\langle do(\alpha_i, s_i) \rangle = \langle \alpha_0, \alpha_1, \ldots, \alpha_i \rangle$$

$$P_f(\alpha, s) = \text{conditions which make } f \text{ take value } v \text{ under action } \alpha$$
$$\neg N_f(\alpha, s) = \text{conditions under which fluent } f \text{ is } \textit{not} \text{ updated when executing } \alpha$$

$$f(s) = \text{value of fluent } f \text{ in the } \textit{Situation Calculus} \text{ for situation } s$$
$$load[f, \langle \alpha_i, \ldots, \alpha_0 \rangle] = \text{value of fluent } f \text{ in the } \textit{fluent store} \text{ after executing actions } \alpha_0, \ldots, \alpha_i$$

where $\langle x \rangle$ indicates an ordered set; in this case the situation is converted to a set of elapsed actions ordered by timestamp.

Theorem: *Jolog* maintains the same fluent values as the Situation Calculus. That is, given a sequence of actions $\alpha_0, \ldots, \alpha_n$, an action theory Σ and a Jolog program δ:

$$\left[\Sigma \models f(do(\alpha_n, do(\alpha_{n-1}, \ldots, do(\alpha_0, s_0) \ldots))) = v\right] \text{ iff } \left[\delta \models load[f, \langle \alpha_0, \ldots, \alpha_n \rangle] = v\right]$$

where v is the current value for fluent f in the respective situation.

Proof: we verify this claim by induction over the length of actions.

Base case: Our base case is when no actions have occurred (the initial situation). Because we initialise the fluent store with all user-specified initial values, we can observe that:

$$v = \text{initial value of fluent } f$$
$$\Sigma \models f(s_0) = v \text{ by definition}$$
$$\delta \models load[f, \langle\rangle] = v \text{ by pseudocode}$$
$$\therefore \left[\Sigma \models f(s_0) = v\right] \text{ iff } \left[\delta \models load[f, \langle\rangle] = v\right]$$

So the theorem is correct in the initial situation.

For the inductive step we assume that the theorem holds for k actions:

$$\left[\Sigma \models f(do(\alpha_{k-1}, do(\ldots, do(\alpha_0, s_0)\ldots))) = v\right] \text{ iff } \left[\delta \models load[f, \langle \alpha_0, \ldots, \alpha_{k-1}\rangle] = v\right]$$

We now prove true for $k+1$ actions. This is done by cases:

Case 1: action α_k does not affect fluent f in situation s_k so $\neg N_f(\alpha_k, s_k)$ holds. Our pseudocode indicates that the fluent store will not be modified, since the particular precondition for fluent f does not hold.

$$\left[\Sigma \models f(do(\alpha_{k-1}, do(\ldots, do(\alpha_0, s_0)\ldots))) = v\right] \text{ iff } \left[\delta \models load[f, \langle \alpha_0, \ldots, \alpha_{k-1}\rangle] = v\right] \text{ by hypothesis}$$

$$\text{however } \left[load[f, \langle \alpha_0, \ldots, \alpha_{k-1}\rangle] = v\right] \equiv \left[load[f, \langle \alpha_0, \ldots, \alpha_k\rangle] = v\right] \text{ by pseudocode}$$

$$\text{similarly } \left[f(do(\alpha_{k-1}, do(\ldots, do(\alpha_0, s_0)\ldots))) = v\right] \equiv f(s_k) = v \text{ by successor state axiom}$$

$$\therefore \left[\Sigma \models f(do(\alpha_k, do(\ldots, do(\alpha_0, s_0)\ldots))) = v\right] \text{ iff } \left[\delta \models load[f, \langle \alpha_0, \ldots, \alpha_k\rangle] = v\right]$$

Case 2: action α_k does affect fluent f, so $P_f(\alpha_k, s_k)$ holds. In this case our pseudocode will ignore the previous value and simply overwrite the fluent with $v' = effect(\alpha_k, s_k)$, thus:

$$\left[\Sigma \models f(do(\alpha_{k-1}, do(\ldots, do(\alpha_0, s_0)\ldots))) = v\right] \text{ iff } \left[\delta \models load[f, \langle \alpha_0, \ldots, \alpha_{k-1}\rangle] = v\right] \text{ by hypothesis}$$

$$\left[\delta \models load[f, \langle \alpha_0, \ldots, \alpha_k\rangle] = v'\right] \equiv effect(\alpha_k, s_k) = v' \text{ by pseudocode}$$

$$\left[\Sigma \models f(do(\alpha_k, do(\ldots, do(\alpha_0, s_0)\ldots))) = v'\right] \equiv effect(\alpha_k, s_k) = v' \text{ by successor state axiom}$$

$$\therefore \left[\Sigma \models f(do(\alpha_k, do(\ldots, do(\alpha_0, s_0)\ldots))) = v'\right] \text{ iff } \left[\delta \models load[f, \langle \alpha_0, \ldots, \alpha_k\rangle] = v'\right]$$

Hence by induction, *Jolog* maintains the situation in a state equivalent to the Situation Calculus.

As a final note, we observe that the Situation Calculus incorporates situations; traditional *Golog*s abstract this from user code while maintaining it internally. *Jolog*, however, legitimately abstracts situations away entirely.

5.4.2 With Backtracking

Out of all Gologs, *Jolog* is closest to *Indigolog*. This is particularly true in regard to actions; an action may have an effect on the world which cannot be undone and accordingly both languages refuse to backtrack actions and hence fluents. However while we may not allow this behaviour, we do present a potential avenue for introducing it to *Jolog*.

First recognise that we still need not remember situations; we just need to ensure we can potentially regress the fluent values. To this end, we extend the fluent store to act as a version-controlled repository of fluent values. Actions have the effect of introducing a new 'version' of fluents while backtracking reverts this value to the previous version (and forgets the top-most version). We make this implementation more efficient by only storing *deltas* between versions (just as a normal version control system would). When looking for the value of a fluent, we consider the most-recent version and then look progressively further back until we find a binding or we run out of versions (in which case the fluent does not exist). Note that we could detect this at compile time for some fluents. However the potential for unknown, dynamically created fluents necessitates at least some detection at runtime.

This approach is sufficient for fluents that do not exist. There is no point in optimising the error case: it shouldn't happen under normal circumstances. However it may prove to be too inefficient for fluents that have not been used recently – constantly reloading them requires digging further and further down the version stack. We propose two alternatives to address this issue:

- *Rolling the database:* we cap the version stack size at some fixed value to prevent it diverging to some unreasonably large depth. If the stack exceeds its limit, then the final value of each fluent is calculated and replaces the initial values. The stack can now be cleared without losing fluent information. Note that this approach prevents unlimited backtracking – the sequence information defining previous states has been lost. This method is adopted in *Indigolog* despite this restriction[5].

- *Caching:* when loading a fluent that is not referenced in the most recent version, we look for its value down the entire stack of versions, as before. We then add this value to the most recent version providing faster repeated access.

We make two small assumptions in this new environment: a) if we regress beyond the first version then the fluent doesn't exist, b) if we backtrack beyond the first version then we have backtracked beyond the initial state and our program should fail.

[5]Interestingly, the deficiencies of this approach are largely cancelled by assumption 1 (above) and so it is rarely a problem.

5.5 Actions

Jolog provides three distinct types of action, each with a specific purpose:

- primitive actions – for general modification of fluents
- exogenous actions – notification of a change in global state, used for updating fluents that define an internal model of the world
- sensing actions – for querying the state of an external agent or system

Primitive actions in *Golog* behave similarly to a normal function call. These semantics have been reproduced in *Jolog*. Exogenous and sensing actions are a little more complicated and are treated slightly differently in *Jolog* to previous Gologs. In particular, exogenous and sensing actions need to interact with external systems – other classes or agents, files, streams, sockets, etc. As such a simple function call is no longer sufficient. Strings present the most intuitive method of allowing this interaction. However this unnecessarily restricts the information that can be shared between agents. To this end we introduce 'messages'. When 'calling' an exogenous action the sender now forwards strings of data – as well as numerical types or serialised classes – to the target agent (possibly itself) through an output stream. A new $jolog.Message$ class has been added to assist this process and to allow additional metadata to be attached, such as a timestamp, information about the sender or more sophisticated features like message ontologies that are required for universal agent communication. The content of this message is then matched against a vector of regular expressions. If one of these patterns matches, then the corresponding action (function) is triggered. Note that *all* *Jolog* actions are functions, but the expected calling conventions vary between the different actions to reflect more common and intuitive usages.

Another distinction is in the relation between exogenous and sensing actions and their return values. These two action types are intended to be related in multiple-agent systems. For example, if $agent_1$ wants to know how $agent_2$ is feeling then it can fire a sensing action targeted at $agent_2$. This action will in turn send a message which is received by the exogenous action handler in $agent_2$. If this matches an exogenous action pattern then $agent_2$ will run the corresponding action body. At this point the exogenous action (may) send a message *back* to $agent_1$. Agents can effectively sense, inform and otherwise communicate under this model.

At this point it may be helpful to cement our definition of return values. Firstly, primitive actions do not return – they are an indication (usually to ourself, though other classes are permitted to call our primitive actions if that is considered useful) that some action should occur, we do not

care what happens subsequently[6]. Exogenous and sensing actions however are linked to streams, so this is the target of their return values. An exogenous action is triggered by a read on the input stream, the return statement is a response to calling this action. It may be as simple as sending "ok." back to the caller to indicate that an action has been successfully updated, or it may be the value of a fluent if this exogenous action is defined like a 'getter' function. Sensing actions behave in the opposite way: they trigger a message by writing to a stream. These actions then wait to read in a response back off this stream. The response can then be used inside the sensing action to update internal fluents. Finally, we confess that this discussion of 'streams' is actually achieved via interaction with our parent 'environment'. The environment acts like a router to our system: it reads in sensing requests and distributes the message to the correct agent via a function call to this agent's *receiveExog* function. This function then chooses which exogenous actions to run following the above procedure. Finally, the result is returned by *receiveExog* to the environment which performs the actual send to the original caller. An analogous process is invoked for initiating sensing requests.

5.6 Non-Deterministic Execution

Prolog provides a powerful backtracking search mechanism as part of its function call semantics – at each 'choice point' the interpreter will select the first matching function definition (from potentially many matching definitions) and run it. If this function succeeds then it returns and execution continues normally. If the function fails for some reason (such as an unsatisfied action precondition, assertion or 'test' ?(*condition*)) then the interpreter will backtrack and choose an alternative function definition in the hope that it is better suited to the current situation. The result is that the interpreter eventually selects a path through 'execution space'[7] such that the program completes successfully.

This is an iconic and often useful feature of Prolog and hence Golog. Accordingly it should be integrated into *Jolog*. This has been achieved with the use of Java exceptions.

[6]Unless the precondition does not hold, in which case the action will throw an exception. However this backtracking is an internal feature – primitive actions do not return as far as user code is concerned.

[7]Provided such as a path exists.

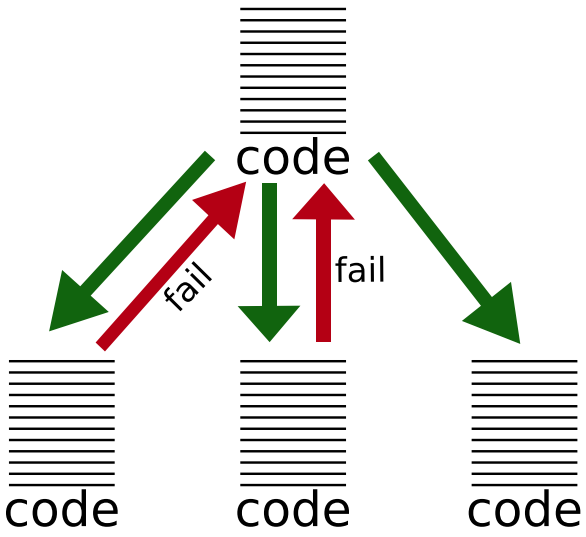

Figure 5.3: *Prolog* backtracking model

Every Jolog function, action and nondeterministic operator (*pick*, *ndet* and *kleene*) threatens to throw a *JologPreconditionException* if it fails for any reason, such as an unsatisfied precondition or a contradiction in program logic. This is complemented with exception handlers around each function and action call and these same nondeterministic operators (*pick*, *ndet* and *kleene*). The result is that a program may call an action that cannot be satisfied, an exception will be thrown and the program will regress to the last choice point – the last exception handler – which will then attempt any secondary actions or functions. If no other options are available then the program will throw a new exception and regress further. This process of progression and regression emulates Golog's Prolog-style function semantics and allows user programs to 'succeed' (reach the end of their program) or 'fail' (throw exceptions and regress beyond the starting point) as we expect.

Note that when producing assembly for exception handlers *Jasmin* accepts a special keyword *any* in place of a normal catch type. In this case the exception table entry will inform the JVM that the type of the thrown exception is not important and to only consider the location in program code that the exception was thrown from. This keyword is used by all *Jolog* exception handlers and allows *Jolog*'s nondeterministic operators to replace the traditional *Java* try-catch statements. For example, if the JVM fails to open a file then it will throw an *IOException* which would normally need to be caught:

Listing 5.5: *Java* code for opening a file

```
1 try {
2     FileInputStream fis = new FileInputStream(new File(filename));
3     ...
4     fis.close();
5 } catch (FileNotFoundException e) {
6     System.out.println(e.getMessage());
7 } catch (IOException e) {
8     System.out.println(e.getMessage());
9 }
```

However, in *Jolog* we can replace this with:

Listing 5.6: valid *Jolog* code for opening a file

```
1 ndet({ FileInputStream fis = new FileInputStream(new File(filename));
2         ...
3         fis.close();
4       } | System.out << "IO Error" << endl; );
5 cut;
```

Note that the nondeterministic operators are not a complete replacement; *Jolog* uses exceptions for control flow and for providing human debuggers with some indication of where failures occur. The above *Java* code gives explicit in-code access to the exception so its error message can be retrieved. This is not currently possible in *Jolog*. Howevermy this sort of functionality is tangential to what *Jolog* is intended for. Accordingly, if this level of expressiveness is required then it is best achieved in a plain *Java* file which can be called natively by *Jolog* code instead.

Performing the backtracking itself is relatively simple: careful definition of exception handler ranges allows the JVM to do most of the control flow for us. This is not the full story though. When backtracking we are actually pretending that we never executed the backtracked code in the first place. This necessitates saving state – which must be done at the start of each choice point[8] – and restoring state – which must be done when backtracking. Fortunately *Java* provides a reasonably convenient mechanism for achieving this via *ObjectStreams*. At each choice point we create a tuple of local variables (an array of *Objects*) and initialise a new *jolog/ChoicePointState* object with it. During initialisation, the *ChoicePointState* object internally creates an *ObjectOutputStream*

[8]The explicit list of operators that require this are *ndet*, *kleene* and *pick*. Function calls are technically choice points, but we delegate the actual saving and restoring of state to any internal operators that define their own choice points.

on top of a *ByteArrayOutputStream* and pushes the input tuple onto the *ObjectOutputStream*. This has the effect of 'serialising'[9] the object so that it can be saved. The choice point object then extracts the underlying Byte array and stores that for later use – this is the saved state in a raw form. The temporary streams are no longer important and can be deleted. Finally, this choice point object is pushed onto a stack stored in the private class variable *jolog_back_stack*.

When restoring state we grab the top *ChoicePointState* off the internal backtrack stack and use its internal Byte array to initialise a new *ByteArrayInputStream*. This in turn is used to create an *ObjectInputStream*. We can then create a new tuple containing saved state variables by reading from this stream (which is simply deserialising the data saved in our internal Byte array). The choice point object then returns this tuple back to our Java code which proceeds to extract individual variables and push them back into their corresponding slots. Note that this process is for local variables only – since fluents cannot be backtracked – and all variables and their variable slots are known at compile time. This allows us to fully generated all support code for these calls to *ChoicePointState* objects and insert it inline with the rest of the user program.

The particular details of serialisation are left to *Java* documentation which is very good on the topic. However we will mention one other feature that *Jolog* recognises; not all objects should be serialised. In fact *Strings* containing sensitive passwords or *Thread* objects that only make sense here and now on the current machine should not be *allowed* to be serialised. Worse, some objects actively refuse to be serialised and will cause the program to fail if serialisation is attempted. To prevent unwanted serialisation *Java* introduces the keyword 'transient' which indicates that a class field should not be serialised. We do not have class fields so we neglect this case, however, we do accept this keyword as a prefix to local variables. A local marked as transient will not be included when saving state at a choice point.

Finally, we confess that we sneakily introduced the *cut* operator in the above code. *Cut* is (in)famous from *Prolog* as the ! operator which 'cuts' choice points away. Essentially it throws away all choice points at the point of execution. If a program fails and backtracks to a *cut* instruction then the program fails. This can be viewed as breaking the purity of the logical representation, however, it is often an essential component of a larger system. In the above case we use it to prevent code from trying to reopen a file which would be a bad thing. We feel that circumstances such as this (or actions which cannot be backtracked) justify its introduction into *Jolog*. The implementation of *cut* is very simple given this: when executed it clears the class

[9]Note that *Java* spells this 'serialization'. This is a process of converting an object – including all of its current state – into a (possibly binary) string for distribution or saving. In fact serialisation allows *Java* objects to be saved to files or sent over a network and then restored – deserialised – and begin running again without loss of information!

backtrack stack (*jolog_back_stack*). It also adds an exception handler to the end of the function: if the cut exception hander is triggered then a 'ProgramFailureException' is generated which terminates the entire program.

5.7 Pick (*pick*), There Exists (*some*) and For All (*all*)

Pick is an uncommon programming construct – it allows the user to request a binding (a value) for a variable such that an affiliated section of code using that variable will succeed if such a value exists. That is, the user can delegate the correct instantiation of a variable to the compiler (or the compiled code in our case). Golog has always left this heavy lifting to the underlying *Prolog* system – a luxury we can no longer appeal to. However an important observation is that *pick* generally boils down to pattern matching against fluents or their values. Our implementation considers it sufficient to enumerate all values of the declared type within the fluent store (including predicate arguments) and store this value into the requested variable. The body of the *pick* can now run with an instantiation preprepared for the target variable. This will either succeed (the 'right' variable was chosen) or it will fail, causing the *pick* to backtrack and attempt a different binding. The inner details of determining variable instantiations – scanning though through the fluent store – are actually implemented in *Java* code, so improvements or corrections are likely to be much easier and less likely to require getting hands dirty with *Jasmin* assembly[10]. We reveal that this process exploits the most recent choice point's 'backtrack id' to achieve enumeration: we increment this number each time we backtrack to the *pick* statement (including if the user program fails). We can thus use this number as an argument to the fluent store, which will return the id$^t h$ value of its target type as the next variable instantiation.

This is not quite the full story though. If the 'correct' variable binding is also a value in the fluent store then it will work, however, the body is arbitrary code which may make no reference to fluents. We endeavour to account for this case by enumerating all values of the specified variable type if it is 'enumerable'. This is of roughly the same standard as *Indigolog* and so we accept some limitations of our *pick* implementation. We are quick to defend ourselves, however, by emphasising that no Golog we are aware of can enumerate integers (let alone real numbers). *Pick* is not a crystal ball that can solve Lagrangians or other complicated mathematical expressions, it is bound by semantic and computational restrictions and so we cannot expect too much of it – nor should we if we abuse its intended usage. In the case of numbers all that any Golog version offers is to enumerate fluent values – and this is only if you carefully structure your program δ. *Jolog*,

[10]Indeed this was the reason why the *fluent store* was written in *Java* code in the first place.

however, will provide this functionality for free (neglecting any performance impacts) and so we accept this implementation as being of an adequate standard.

Note that we formerly endeavoured to account for additional tricky cases by providing a special 'variable' class. This class was implemented to behave similarly to *Prolog* variables – it is initially 'unbound' and binds to the first value it is compared against. In most cases this variable would be bound internally to a value in the fluent store before running the user program, however, for the first attempt of the *pick* statement we would provide an entirely unbound variable. This means that the first use of the variable within the body would bind its value. This method suffered from two crippling disabilities: it would have severely complicated the entire codebase, and worse, it would not reliably buy us anything we don't already have in the previous implementation.

Finally, while discussing *pick* we should indicate how enumeration of types is possible. Indeed at this point we must recognise that *Java* as a runtime system provides some amazing features, in this case it is even able to emulate type enumeration. This can be trivially achieved in *Prolog* by wrapping a 'type' inside a predicate and backtracking though the list, however, it turns out that *Java* is perhaps even more convenient:

Listing 5.7: simple *Java* 'Reflections' code to enumerate an 'Enumerable' object

```
1  Class.forName("java.util.concurrent.TimeUnit");
2  if (type.isEnum()) {
3      for (Object c : type.getEnumConstants()) {
4          System.out.println(c);
5      }
6  }
```

With an implementation of *pick* working we can now utilise this construct to provide quantifiers: the *some* and *all* constructs are both defined in terms of *pick* as follows:

$$\text{some(type } t \mid \phi) \stackrel{def}{=} \text{ndet(pick(type } t) \{\ ?(\phi); \text{ return true; } \} \mid \text{return false;)}$$

$$\text{all(type } t \mid \phi) \stackrel{def}{=} \neg\ \text{some(type } t \mid \neg\phi)$$

Note that these constructs are compiled inline, so 'return' is actually a 'load' instruction. The use of return above is a little closer to the operation of this construct as a system, since we know that *ndet* will leave this value on the stack. We do not recommend exploiting this trick within user code though.

5.8 Nondeterministic Branch (*ndet*)

The implementation of *ndet* appears reasonably convoluted at first, however, it operates on a few simple rules that conspire to provide the semantics we expect.

Listing 5.8: code template for nondeterministic branch (*ndet*)

```
1  save initial state
2
3  doNdet1:
4     δ1
5     save dummy state (no variables) with backtrack id = 1
6     goto endNdet
7
8  redoNdet1:
9     throw exception //backtrack back into δ1
10
11 doNdet2:
12    restore top state (should be initial state)
13    δ2
14    saveBacktrackId(2)
15    goto endNdet
16
17 redoNdet2:
18    throw exception //backtrack into δ2
19
20 handler: //backtracking exception handler
21    get backtrack index from top (dummy) state
22    pop this dummy state and throw it away
23    if (backtrack index == 1) {
24       goto redoNdet1
25    } else if (backtrack index == 2) {
26       goto redoNdet2
27    }
28
29 failPoint:
```

```
30      pop state
31      throw exception //backtrack
32
33  endNdet:
```

with the corresponding exception table:

from	to	with
doNdet1	doNdet2	doNdet2
doNdet2	handler	failPoint
endNdet	endScope	handler

A critical element of *ndet* is its treatment of the backtrack stack. All nondeterministic *Jolog* operators are required to keep out of each other's way. An important step towards achieving this is for each construct to 'clean up' after itself on failure – if an *ndet* subprogram fails then it needs to restore the backtrack stack to the state it was in before the *ndet* was called. We will now examine an execution of *ndet* to simultaneously familiarise the construct and to verify its treatment of the backtrack stack is satisfactory.

Let us assume that the backtrack stack contains γ immediately before calling the above *ndet* code. We can see that when *ndet* starts it immediately saves the initial state. This is another critical step since it allows *ndet* to restore the initial condition before progressing to alternative branches (specifically δ_2). The backtrack stack now contains $\chi_0\gamma$ (where the left most value is the top of the stack and χ_0 is the state prior to executing *ndet*[11]).

Control then flows to δ_1. Let us first consider the case where this program succeeds. δ_1 is an arbitrary program so it may also contain nondeterministic operators. Our backtrack stack may now appear as $\delta_1\chi_0\gamma$. The *ndet* code now resumes and stores a 'dummy' choice point on the backtrack stack – a choice point which stores no local variables. The purpose of this entry is not to save state but to save the backtrack id; each choice point stores a byte array of state information and a 'backtrack id' which we use to indicate the *ndet* branch we are currently exploring. The backtrack stack now contains $\chi_1\delta_1\chi_0\gamma$. We now execute a *goto* and complete the *ndet*.

Future code may add and remove things from the backtrack stack, however, if it fails and backtracks to this *ndet* then we assume the stack is identical to the way we left it. That is, we assume that future nondeterministic operators have cleaned up after themselves, which should ensure the stack appears unchanged. Any future backtracking exception will be caught with our

[11]Note that we should always save the initial state: γ may be empty or there may be other state-modifying instructions between here and the last nondeterministic operation. In either of these cases we will restore an old and inaccurate state unless we explicitly save a copy ourself.

handler, which proceeds to pop the top of the stack χ_1 – our dummy state – and retrieve the backtrack id 'one'. This is an index into our handler vector which indicates we should retry δ_1. We now jump to $redoNdet1$ where we immediately throw a new exception. This exception will either be caught by δ_1 or if that fails (either because it has no nondeterministic operators or because it has exhausted all of its nondeterministic options) then control is caught by the *ndet* itself and handed to δ_2.

We note that immediately after throwing this exception the backtrack stack is $\delta_1\chi_0\gamma$ – that is we have restored the backtrack stack to the state that δ_1 left it. If we assume that this program is implemented correctly then it now has the state it expects and can decide how it wants to handle failure. This program may find an alternate execution, succeed and modify the stack. We will then reappend our dummy state χ_1 and execution will continue after the *ndet*. This may happen as many times as δ_1 can support – potentially infinitely in the case of the $Kleene$ operator. However if δ_1 should fail then the first *ndet* exception entry will catch the exception and hand control to δ_2. We note that for δ_1 to fail it must have cleaned up the stack to the point at which it was called. The backtrack stack must then contain $\chi_0\gamma$.

At the point we are about to attempt an alternate branch of the *ndet* as if we had never executed δ_1 in the first place. To do this we must first restore the fluents to the *ndet*'s initial state, which is conveniently at the top of the stack. Execution now continues as before with two distinctions: we execute δ_2 now instead of δ_1 and our dummy state is now χ_2 – the second branch corresponds to the backtrack id of two. This branch is also complemented by an entry in the exception table and a corresponding handler in the *ndet* code. This should be sufficient evidence that the second branch will behave equivalently to the first.

The remaining case is when δ_2 fails, which indicates that the entire *ndet* has failed. In this case the exception is fired between the labels $doNdet2$ and *handler* so the second exception entry will match and pass control to the *ndet* fail point. We assume that δ_2 is implemented correctly, so on failure it restores the backtrack stack. Thus the stack contains exactly $\chi_0\gamma$. However χ_0 was the situation immediately before the *ndet* which is of no use to anyone but us, so we throw this entry away. The backtrack stack is now at γ – we have successfully cleaned up after ourself and can safely throw an exception to pass control to an earlier choice point.

Note that in our execution we never got in the way of another (well-behaved) nondeterministic operator – potentially including our subprograms. This design was chosen for its simplicity and its flexibility; we can now construct arbitrarily nested nondeterministic operators:

$$ndet(ndet(\alpha|\beta) \mid ndet(\gamma|\delta))$$

5.9 Nondeterministic Iteration (*Kleene star*)

The nondeterministic iteration operator – the *Kleene* star – behaves in a similar way to *ndet*. However we now have the luxury of ignoring the backtrack id – when we backtrack to the *Kleene* construct we simply restore our saved state and throw the save away. Next we run the user program one more time and save the resulting state as if it is our initial state.

Listing 5.9: code template for nondeterministic iteration (*Kleene star*)

```
1     save initial state
2
3  //try zero executions first ...
4     goto endKleene
5
6  //do the program once more ...
7  startKleene:
8     restore state
9     throw our save away
10
11    δ //run the user program
12
13    save state
14
15 endKleene:
```

The above code is complemented with the following exception table:

from	to	with
endKleene	endScope	startKleene

Note that *Kleene* does not have a 'fail point' like *ndet* does. The beauty of this implementation is that we just run the user program one more time and save the resulting state for the next round. If δ should fail then it will tidy itself up and throw an exception. We have already fullfilled our failure conditions at this point though so *Kleene* can let an earlier construct catch the exception without intervention.

5.10 Function Calls

Function calls are very simple in *Java*: push the object reference (for non-static functions), then push all arguments in order, then invoke the desired function with *invokevirtual* (or *invokestatic* or *invokespecial* for static methods or constructors respectively). The issue however is that *Jolog* is required to backtrack to previous choice points – *which may be inside a function that has returned*. The adopted solution to this problem is to wrap a function call in a 'choice point-like' exception handler structure. That is, when calling a function for the first time we do the normal method above, but the compiler also supplies an unnamed Boolean value 'false'. The target function will then run, *possibly adding its own choice points to the backtrack stack*, then return. Execution can now continue with the rest of the program. If a later failure forces backtracking to this function call then we catch it and attempt to rerun the function. However we now supply the value 'true' as this final parameter.

We now examine how a function interprets the value of this mysterious parameter. The first line of each function is not user code – it is a special check against the value of this internal parameter. If the value is false (this is the first call) then execution continues on to user code, however, if the parameter is true then we execute a jump to the very end of the function (after all user code, including the function return). At this point the function throws a backtracking exception. There are now two potential results: the function does not contain any choice points (nondeterministic operators or function calls) or it does. In the case of no choice points then there will be no exception handlers so the function will immediately fail. This will backtrack to our initial function call and then attempt to be caught at this point – it is as if we never called the function at all.

The other case is if the function *does* have choice points. We first note that the function call has not touched the backtrack stack (in fact it never does). This means that before the first call to this function we have a backtrack stack of γ. The first (successful) completion of the function will leave the backtrack stack at $\delta\gamma$ (since we assume here it has at least one choice point). Execution continues in the base function then backtracks to this point. Our enforced semantics for nondeterministic operators state that we should find the backtrack stack the way we left it, so it should still be $\delta\gamma$. Calling the function again and jumping to the end will not change this. At this point we throw an exception which must be caught since the function has a choice point in it. Further the backtrack stack is correct for its purposes. Control will now continue as in normal execution and we will run the base function again, or if we have run out of valid choice points in this function then it will backtrack – the function will fail with an uncaught exception. This case behaves identically to a function call without any choice points as explained earlier.

The various potential execution paths are summarised in the following diagram:

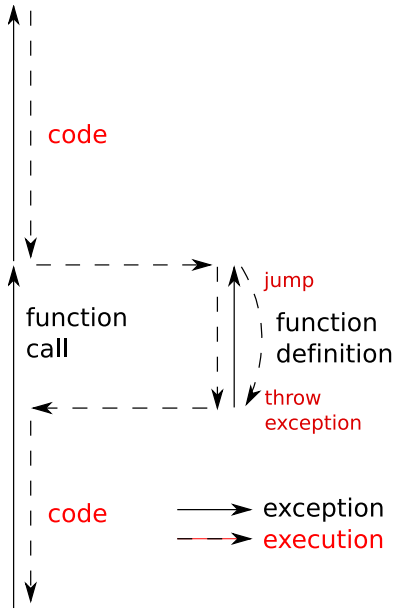

Figure 5.4: Competing control flows for *Jolog* functions

We now present the code template for function calls:

Listing 5.10: code template for *Jolog* function calls

```
1     fn(x⃗, false)
2     goto end
3  retry:
4     fn(x⃗, true)
5  end:
```

with an exception entry:

from	to	with
end	endScope	retry

As a final note we observe that this backtracking mechanism makes sense in a 'linear' function. A function with multiple exit points requires the backtrack to start at the *correct* return. This is tricky (though it could be achieved in an analogous manner to *ndet*) and has no real impact on the expressiveness of a function. Accordingly we disallow multiple returns. This feels more appropriate since it is more 'structured', and more importantly, *Prolog* (and all Gologs built on top) behave in the same manner – they have no notion of a return value in the first place.

5.11 Tuples

Emitting a tuple is relatively straightforward; in all cases, a tuple used as an 'rvalue' (as a constant to the right of an assignment operator or as an argument to a function) is most useful as a normal array – its reference can be assigned to a variable or passed to a function in one operation. To this end, tuples are emitted by first creating a new array, then emitting each component expression and assigning to the relevant array cell.

Assignment to a tuple is a trickier process. First we want to ensure that only valid (left hand side, or 'lvalue') tuples can be assigned to. So we can say, for example:

$$(x, y) = (y, x)$$

for fluents or local variables x and y, but we cannot write:

$$(1, myFunction()) = (x, y)$$

since assigning to the integer literal 1 or the function $myFunction()$ does not make sense. This sanity check is achieved by delegating the assignment to each of the individual components: a tuple object within the compiler stores a vector of expressions. When assigning we simply assign to each expression in turn. If the tuple we assign to stores fluents and local variables, then each one of these will be assigned correctly (and perform the necessary type checks). If the tuple on the left of the assignment operator, however, is a literal or a function call then these objects will offer their own error messages as to why the tuple assignment cannot proceed.

Note that in the above (legal) example the syntax requests two tuples for an assignment, where the tuple to the right hand side loads the values of x and y into an array. The left side, however, should *not* create a tuple – we want to assign the *values* on the right into the corresponding *variables* on the left. In fact, this example utilises tuples to avoid (explicitly) creating a temporary variable when swapping two values. This is a canonical example of Python tuples and accordingly serves as a standard for our own implementation.

It should be evident that our approach for emitting (evaluating the right hand tuple and storing into an array) and assignment (iterating through each element and delegating the assignment to the corresponding expression) both work under these additional restrictions. The catch in this arrangement comes from loading values onto the execution stack. When assigning to a tuple of local variables we can simply emit the complete initialisation expression to get an array or tuple, then duplicate this and assign to each corresponding element in our lvalue tuple. This approach no longer works if we are assigning to a tuple of fluents, since fluent assignment requires a $fluent\_store$

object on the stack. That is, we need prefix and postfix operations for each element of the initialisation expression. The current mechanism for assignment mandates that expressions that support assignment receive a single initialisation expression and deal with everything else internally. This produces a neat interface that is sufficient for all other constructs. However, the convenient abstraction becomes a problem in this case, since it obstructs division of the initialisation expression.

There are at least three potential solutions to this problem without affecting the existing interface:

- Re-emit the initialisation expression for each individual assignment and pick out the target component
- Emit the initialiser once, emit the assignment prefix and do complicated stack reordering
- Emit the initialiser once, store into a local variable and retrieve components from this

The first proposal has two severe drawbacks: first, it is much less efficient due to duplicate code, and second, if the right-hand tuple includes function calls with side effects (file IO for example) then we could easily produce unexpected results.

The second suggestion is worse since it will not always work: local variables are still fine, but fluents can potentially require two prefix values on the stack before each instance of the initialiser. The bytecode instructions are quite expressive, allowing duplication as well as direct and indirect swapping, but they prevent distant stack manipulation. We could restrict tuple assignment to local variables, but this is inelegant and violates our self-imposed abstraction.

The third proposal does not suffer from any of these restrictions – it allows the initialiser to be emitted once, stored, and its values retrived internally without breaking abstraction or making any assumptions about stack usage. The merits of this approach clearly justify its implementation.

Chapter 6

Syntax

6.1 Type Checking and Name Resolution

The scope of Jolog has grown from its initial conception of a small, standalone language, to a larger beast capable of running arbitrary Java code. While this has enhanced Jolog's usefulness, it has also introduced several unique issues. This is especially true in regards to type checking. Most modern languages are able to differentiate between variables, functions and values fairly easily due to distinct syntactic features and lookups in a symbol table. This is not so trivial in Jolog since there is an expectation to preserve duplication in syntax and in language constructs available in previous versions of Golog.

In particular, we wanted Jolog's syntax to exhibit the following properties:

- Fluent names should be predicates, keeping with Golog's logic theme
- Pick statements should be able to bind variables to fluent arguments *and* values
- Dynamic fluents should be permitted (that is, we do not know at compile time whether a fluent exists)
- Functions and actions are essentially the same creature to the end user, so their calling syntax should be related
- Disambiguation via spurious prefixes and suffixes should be avoided

These constraints highlight several issues:

- References to fluents, functions and actions all have the same form

- Fluent names and values cannot necessarily be distinguished and resolved to code at compile time: they may both be predicates and the compiler may have never seen either declared

All flavours of Golog to this date have avoided these issues by relying on sensible users and *Prolog*'s pattern matching to pick something that might work. However this approach is unsatisfactory in *Jolog* since it would mandate a ridiculous implementation and it erodes the notion of types that we want to introduce and enforce.

The adopted solution is achieved via our definition of *Jolog* name resolution:

- Local variable names must be atoms[1] (as in *C* or *Java*)
- Fluent names must be predicates or atoms
- Function names (including primitive and sensing actions) must be predicates
- Fluent values are 'encouraged' to be tuples or any non-predicate type
- Actions are compiled to be identical to normal functions and are treated as such internally
- Primitive and sensing actions are called like normal functions
- Exogenous actions are usually called externally and work on message passing, so these are called via new communication primitives
- A predicate will be resolved to a function call if a matching function definition (including a primitive or sensing action) has been registered, otherwise it will be treated as a fluent

Note that the restriction on fluent types is at compile time and applies to name resolution only: the *Jolog* compiler has to decide what Java bytecode to generate. In particular, there are different operations for fluents and local variables – this heuristic means the compiler can assume that if it hasn't seen a predicate before, then it is just a dynamic fluent and fluent code should be generated. All other types will then be treated as local variables, literals or function calls based on their own distinguishing syntactic features and the above resolution rules. There is no restriction against storing a predicate in a fluent though – fluents can store arbitrary types – so a less direct approach will achieve these ends, for example:

[1] An atom is a predicate without arguments; in our case it should not include parentheses either.

Listing 6.1: valid *Jolog* code for storing a Predicate as a fluent value

```
1 //manually create a predicate: head(3)
2 Predicate p = new Predicate("head");
3 p.addArgument(3);
4
5 //store this predicate into fluent: f("my fluent")
6 f("my fluent") = p;
```

That is, the compiler won't hide the process of creating the predicate – this brand of syntactic sugar is for fluent names only.

6.2 Agent

Jolog is an agent-oriented language, so Java classes are naturally replaced with 'agents' – extensions of regular classes. In particular, an agent allows (non-static) functions, local variables and class 'fluents' but complements these with actions: a mechanism for communication with other agents and controlled modification of fluents. Note that static functions are still permitted (in fact, a valid *main* function can be written in *Jolog*). However these functions have severe restrictions on their functionality, in particular they cannot contain any nondeterministic constructs (since there is no access to the class' 'backtrack stack'). Note also that a fluent's value is accessible throughout an agent, but can only be modified by an action – as in *Indigolog*. This restriction is justified by the powerful operations available only to fluents, namely the pick and search features.

Besides the 'agent' prefix, an agent is syntactically identical to a *Java* class:

Listing 6.2: *Jolog* agent template

```
1 agent agentName {
2     fluents
3     actions
4     functions
5 }
```

where fluent, action and function definitions may appear in any order the user may desire. This shuffling of definitions is possible since compilation detects all function and action prototypes first, then proceeds to compile their individual bodies. This two-step process allows us to resolve all function and action names inside any function or action body without the need for forward declarations. In fact this 'feature' is essential for our resolution system to correctly disambiguate between fluents, actions and functions which all share identical syntax. The specifics of these components are discussed in the remainder of this chapter.

6.3 Fluents

Modern Golog variants use 'fluents' – constructs more familiar as local variables – and so this pattern is continued in *Jolog*. A major distinction between fluents and variables is in the increased functionality that fluents provide:

- dynamic insertion and removal
- pick (non-deterministic selection/instantiation)
- search

The complexity introduced by these features means they are more conveniently implemented in Java code – rather than composed in assembly by hand – and so code to load, store and reason with fluents in the fluent store is written in pure Java. Accordingly only the final result needs to be dealt with in hand-written assembly. This has the advantage of being more reliable and easier to change. It also reduces the size of agent files since fluent code appears once, and this is in the external fluent store object. Agents can now call functions in the fluent store rather than duplicating these common routines.

Fluents are characterised in the language as predicates:

$$name(arg1, arg2, \ldots)$$

where the name is an identifier (an underscore or alphabetic first character, followed by an arbitrary number of underscores and alphanumeric characters). For example:

Listing 6.3: examples of *Jolog* fluent definitions

```
1 int fluent;
2 Object anotherFluent(0, "second argument") = new Object();
```

The value of a fluent has very few restrictions; it is of type java/lang/Object. This means it can hold any reference type, including arrays, but no primitive types. Care has been made to ensure that the Jolog compiler implicitly converts primitive types to their corresponding reference types when storing, and converts back when loading, so that fluents of primitive type should appear identical in Jolog code.

The only other major restriction on fluent values is that they should not ordinarily be of type *jolog.Predicate*. This class is used for fluent names and the distinction is drawn to ease in compile-time resolution of unknown dynamic fluents, as discussed above.

6.4 Message Passing and Exogenous Actions

Jolog aims to introduce Golog to a more 'distributed' realm, and personal experience indicates that message passing is a preferable form of communication – in terms of simplicity, correctness and provability – under this domain (it is certainly superior to shared variable solutions in all but simple cases). Some convenient syntactic sugar is borrowed from *C++* to encourage this paradigm: input from an InputStream (or subclass) can be read via the >> operator:

Listing 6.4: valid *Jolog* code for reading a *Message* from an *InputStream*

```
1 InputStream input;
2 Message m;
3 input >> m;
```

Similarly, *Messages* can be sent with the << operator. Note that these operators are not restricted to sending *Message*s: any (serializable) type can be sent, including reference types like *String*s and primitive types like *int*s or *float*s.

Note also that *System.in*, *System.out* and *System.err* are *InputStream*s and *OutputStream*s – as are user files – so convenient *C++*-style I/O can be performed using these operators.

Exogenous actions can now be viewed in terms of these operators. If the 'world' changes, then it is the 'world' agent who is responsible for informing all its citizens of the change. Further, because the world agent may be an external system (a separate program, hardware, user input, etc) we need to be informed via a general mechanism: *Message*s over *InputStream*s. Internally an exogenous action is caught and redistributed by an agent's 'environment', which forwards the *Message* to each agent by calling their *receiveExogenous(Message)* function. Receiving exogenous actions is accordingly transparent to user code. However if *we* are the world agent then we need some way of signalling a change – we need to *send* an exogenous *Message* to an environment (or to a specific agent if we so desire). This is achieved with identical syntax to above:

Listing 6.5: valid *Jolog* code for sending a *Message* to an *InputStream*

```
1 Environment myFriends = ...;
2 myFriends << "hello, my favourite number is " << 9;
```

Note that data to be sent will be combined into a single *Message*, so that the *myFriends* environment will receive "hello, my favourite number is 9".

The specific implementation of exogenous actions warrants its own discussion; the template for an individual action is provided here:

Listing 6.6: code template for individual exogenous actions

```
1 Message exogenousAction_i(Message message) {
2     if message matches pattern i:
3         update fluents using successor state axioms
4         return new message
5     else:
6         return null
7 }
```

Firstly it should be noted that the first line performs a *Java* regular expression match with the input message. This simply establishes that we are executing the correct function for the given action – it is an implementation detail to allow execution of multiple actions with arbitrary 'names' where each action has to check that its pattern (as opposed to its name) matches the input message. If the message does match then we check the precondition ϕ. Note that if the message action does *not* match the action pattern, then the action handler backtracks (throws an exception) and attempts a different action such that we only execute relevant (user) code for the correct actions. In this way the correct action can be 'chosen' and executed at runtime.

Secondly, no exogenous actions have preconditions (or equivalently, they all have *true* preconditions). Exogenous actions are otherwise equivalent to primitive actions[2].

We can argue from these two points that the implementation is the same as in *Congolog*:

$$\delta_{EXO} \equiv (\pi a \ |?(isExogenous(a)); a)^*$$

Note that the correctness of this claim relies on the implementation of primitive actions – as in *Congolog* – which is discussed later. We let this suffice as a proof of correctness: there are no definitions of trans or final relations for δ_{EXO} because this program is not directly accessible within user code.

6.5 Functions and Expressions

Care has been taken to ensure that Jolog expressions, function calls and function definitions appear just as in normal Java code. Two exceptions are the absence of bitwise operators (these are used solely for actions/IO) and the absence of generic type parameters (enclosed in angle brackets). Note that Java generics are actually just clever inference by the compiler: the actual type information is

[2]In fact much of the code to generate these actions is common.

discarded and all generic types are stored as *java/lang/Object*. This design decision makes generic code backwards compatible with previous *Java* bytecode versions without restricting modern *Java* compilers from reasonable type safety. It also means that the *Jolog* compiler can legitimately ignore these parameters at the expense of some type safety.

The syntax for relational operators has also been slightly augmented to allow a collection of relational (and equality) tests. For example, this more 'logical' notation permits:

$$0 < 1 \leq 2 > 1$$

which returns true. Compiling code like this in a *C*-like language is liable to yield false as each binary relation compiles to a truth result which is used in subsequent calculations. In reality *Jolog* is simply providing syntactic sugar for the more familiar:

$$0 < 1 \ \wedge \ 1 \ \leq 2 \ \wedge \ 2 > 1$$

For completeness, function definitions have a template resembling:

Listing 6.7: code template for *Jolog* function definition

```
1 [privacy] returnType functionName(type_0 argument_0, type_1 argument_1, ...) {
2     //body
3 }
```

where 'privacy' indicates an (optional) access control level of *public*, *private* or *protected* and the parentheses contain a comma-separated list of parameters – a type followed by an identifier[3].

The body of a function contains zero or more 'statements' which include:

- return statement: $return\ [value]$;
- local variable definition: $type\ name\ [\ =\ \phi\ ]$;
- if statement: $if\ (\phi_1)\ \{\ \delta_1\ \}\ [\ else\ if\ (\phi_2)\ \{\ \delta_2\ \}\ \ldots\ [\ else\ \{\ \delta_n\ \}\ ]\ldots]$
- while statement: $while\ (\phi)\ \{\ \delta\ \}$
- for statement: $for\ ([\delta_{\text{init}}]\ ;\ \phi\ ;\ [\delta_{\text{update}}])\ \{\ \delta\ \}$
- test: $?(\phi)$;

[3] *Jolog*'s definition of identifiers is comparable to many other languages: a human readable 'word' that does not start with a digit and consists of only alphanumeric characters and underscores.

- nondeterministic branch: $ndet(\delta_1 \mid \delta_2)$
- nondeterministic iteration: δ^* ;
- nondeterministic choice of argument: $pick(type\ \ identifier)\ \{\ \delta\ \}$
- scope: $\{\delta_1 \ldots \delta_n\}$
- expression statement: $[expression]$;[4]

Void functions (procedures) are issued with a compiler-provided *return* statement if the user chooses to omit it. However the *Jolog* compiler is unable to provide a sensible default value for functions that return a type. Non-void functions that do not specify a return (or specify an incorrect return type) will be treated as errors. Note also that the implementation of backtracking in functions means that nondeterminism becomes frighteningly difficult in the presence of multiple exit points. Multiple returns are accordingly prohibited in *Jolog* functions.

6.6 *pick*, *some* and *all*

Pick allows a user to specify a variable name and a subsequent program that uses a 'nondeterministically chosen' binding of this variable to positive effect. With the introduction of types, we augment this picture by requiring a type declaration too. The result is code that looks like:

$$pick\ (type\ t)\ \{\ \delta\ \}$$

where the user program δ (generally) uses a variable called t of type *type*. Note that *pick* essentially supplies a value to the user program and sees if it works. If δ always fails (or doesn't even use the *pick*'d variable) then *pick* will be powerless to successfully guide the program to completion, as in:

$$pick\ (int\ i)\ \{fail;\}$$

Clearly the failure of this program is unrelated to the efforts of this construct. In fact, in this case *pick* will retry several bindings of i in the hopes that one of them might work. This may be viewed suspiciously although this particular program is fairly ridiculous anyway. Of more use is a

[4]The useless sounding 'expression statement' precludes its true use: the common function call is a particularly ubiquitous expression statement.

program that exploits this implementation to achieve iteration:

$$ndet\ (\ pick\ (int\ i)\ \{\ System.out\ <<\ myFluent(i)\ <<\ endl;\ fail;\ \}\ |\ \{\}\ )$$

Programs like this highlight the convenience of the *pick* construct, which provides the most effective access to an arbitrary fluent, especially when the structure of the fluent's name is known, but the values are not.

While *Pick* deals with specific values, its more 'logical' counterparts simply deal with the 'existence' of values: *some* and *all* correspond to the logical quantifiers $\exists$ and $\forall$. As such, a distinct logical notation is provided for these constructs:

$$some\ (type\ t\ |\ \phi)$$

and

$$all\ (type\ t\ |\ \phi)$$

Note that ϕ is similar to the user program δ in *pick*, however, ϕ is expected to be a Boolean formula. Note also that these constructs are Boolean formulae themselves, so they cannot just appear in *Jolog* programs unaccompanied. Two traditional usages are:

Listing 6.8: Uses of *some* and *all* in *Jolog* programs

```
1 if (some(String s | person(s))) {
2     //we know at least one person exists
3 } else {
4     //no one around ...
5 }
6
7 //make sure no one is rude
8 ?(all(Person p | !rude(p)));
```

6.7 Tuples

Tuples have been introduced to the grammar to provide easier disambiguation between fluent names and values. They are also a popular construct found in the Python scripting language, so their use for storing a collection of data is widely accepted – arguably more so than for a Prolog-style predicate. Further, this introduction can be performed without affecting the expressiveness: a predicate with arity n can be viewed as a tuple of arity $n+1$ with the tuple's first value equivalent to the predicate head. Tuples appear in code enclosed in parentheses:

$$(\nu_1, \nu_2, \ldots, \nu_n)$$

Note that the arity of a tuple must be greater than one; there is no way of distinguishing a single element tuple from a bracketed expression. Even Python breaks this problem with an inelegant syntactic exception[5]. Alternatively, the grammar could be modified to use different syntax, angle brackets for example. In fact this change would be very quick since the only problem is with resolution: the internal code already supports arbitrary-size tuples and the grammar would require only three new characters. However this change would be cosmetic only, and parentheses feel more appropriate. Furthermore the reasons for *wanting* a single-arity tuple are unclear, especially since *Jolog* permits arbitrary types; a tuple of size one doesn't buy anything new.

[5]For those with particular interest, a *Python* tuple of size one can be created with a trailing comma: $myTuple = value$,

Chapter 7

Jolog Implements the Situation Calculus

The process of ensuring that a compiler generates 'correct' code is hard, let alone ensuring all the support libraries also work. Further, the effort spent verifying often far outweighs the potential benefits: a well tested compiler is quicker and easier to produce than a verified compiler and is sufficient for many non-critical systems. Indeed, depending on the proof method, testing can even *strengthen* a proof by ensuring that no shortcuts were taken along the way (when reducing to a simpler model or discharging proof obligations with weak arguments). Very few compilers are verified for these reasons amongst others.

What follows is not a proof of correctness in these terms; *Jolog* does not belong on the Space Shuttle or in a real-time system – its nondeterministic features are not appropriate for these domains and the time delays they introduce are unacceptable, let alone near-impossible to prove properties of. However this does not mean that there is nothing useful we can say about *Jolog*. In this chapter we introduce a proof of correctness not by the above standard, but in terms of something more useful to our domain. Specifically, the following sections develop an intuitive argument for why *Jolog* implements the Situation Calculus – a formal theory of actions – and inherits a heritage in reasoning with first-order logic along the way.

Proof proceeds by showing that the compilation and execution technique described in Chapter 5 satisfies the $Trans$ and $Final$ semantics introduced by *De Giacomo*, et al. for *Congolog*.

7.1 Congolog

The definitions of *Trans* and *Final* predicates as provided in *Congolog*[2] are as follows:

Empty program:

$$Trans(nil, s, \delta', s') \equiv False$$

$$Final(nil, s) \equiv True$$

Primitive actions:

$$Trans(\alpha, s, \delta', s') \equiv Poss(\alpha[s], s) \wedge \delta' = nil \wedge s' = do(\alpha, s)$$

$$Final(\alpha, s) \equiv False$$

Test:

$$Trans(?(\phi), s, \delta', s') \equiv \phi[s] \wedge \delta' = nil \wedge s' = s$$

$$Final(?(\phi), s) \equiv False$$

Sequence:

$$Trans(\delta_1; \delta_2, s, \delta', s') \equiv \exists \gamma \mid (\delta' = (\gamma; \delta_2) \wedge Trans(\delta_1, s, \gamma, s')) \vee (Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s'))$$

$$Final(\delta_1; \delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)$$

Non-deterministic branch (ndet):

$$Trans(ndet(\delta_1|\delta_2), s, \delta', s') \equiv Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s')$$

$$Final(ndet(\delta_1|\delta_2), s) \equiv Final(\delta_1, s) \vee Final(\delta_2, s)$$

Non-deterministic choice of argument (pick):

$$Trans(pi(v)\{\delta\}, s, \delta', s') \equiv \exists \vec{x} \mid Trans(\delta^v_{\vec{x}}, s, \delta', s')$$

$$Final(pi(v)\{\delta\}, s) \equiv \exists \vec{x} \mid Final(\delta^v_{\vec{x}}, s)$$

Non-deterministic iteration (Kleene star):

$$Trans(\delta^*, s, \delta', s') \equiv \exists \gamma \,|\, (\delta' = \gamma; \delta^*) \wedge Trans(\delta, s, \gamma, s')$$

$$Final(\delta^*, s) \equiv True$$

7.2 Assumptions

We start by declaring a few assumptions about our implementation:

- The closed-world assumption and the common-sense principle of inertia both hold.
- All code generated by the *Jolog* compiler is syntactically valid, so a Java Virtual Machine will not crash mid program. This is further restricted by assumptions about user activity: we assume that a given *Jolog* program will not throw any casting errors or *NullPointerExceptions*.
- A compiled Boolean formula has the same semantics as a corresponding Boolean formula, although we augment this with 'short-circuit' conjunctions and disjunctions[1].
- A Boolean formula (denoted here by ϕ) does not contain function or action calls[2]

[1]Short-circuiting *and*s and *or*s is familiar in *C*-like languages, for example: $false \wedge x$ is false, irrespective of the value of x. Short-circuiting is a process of aborting compilation early if we already know the result. In this case it indicates that we do not evaluate x.

[2]Note that *Jolog* allows this functionality if it is truly desired although we make no guarantees about its usage in circumstances where a plain Boolean formula is required. In some cases this may lead to duplicate execution of code and so we legitimately omit these cases from our proof.

7.3 Result

The following result formalises the correctness of a *Jolog* program:

Theorem: Given a *Jolog* program δ and initial situation s_0, the following holds:

$$Trans_{\text{jolog}}(\delta, s, \delta', s') \equiv Trans(\delta, s, \delta', s')$$

and

$$Final_{\text{jolog}}(\delta, s) \equiv Final(\delta, s)$$

Proof: We prove this result by considering each of the basic constructs of a *Jolog* program in turn.

7.3.1 Empty program

In the case of an empty program, the *Jolog* compiler produces empty functions: no code. Providing the Java Virtual Machine (JVM) with no code will clearly result in no code being executed, so the system will be unable to transition the program or the current situation – there are no instructions, so no way of modifying the situation:

$$Trans_{\text{jolog}}(nil, s, \delta', s') \Rightarrow False$$

Alternatively, consider the case where the virtual machine is unable to transition. By assumption 1 above we can then assume that the only reason that the JVM would *not* execute code is if there is no code available to be executed:

$$Trans_{\text{jolog}}(nil, s, \delta', s') \Leftarrow False$$

Furthermore, if the JVM is unable to transition, then it is already finished – it would have executed the last instruction *then* finished, before attempting a non-existent empty program. That is, the program is trivially *Final* in this situation.

$$Final_{\text{jolog}}(nil, s) \Rightarrow True$$

Similarly, if the program has finished – it is 'executing' the empty program – then $Final$ must be true. If the JVM is not finished then it must be executing code and so the program cannot be

empty. We can then assert that:

$$Final_{\text{jolog}}(nil, s) \Leftarrow True$$

Hence, *Jolog*'s execution of the empty program is equivalent to that specified by *Congolog*:

$$Trans_{\text{jolog}}(nil, s, \delta', s') \equiv Trans(nil, s, \delta', s')$$

$$Final_{\text{jolog}}(nil, s) \equiv Final(nil, s)$$

7.3.2 Primitive actions

The *Jolog* code produces *Java* bytecode equivalent to:

Listing 7.1: code template for primitive actions

```
1 void name(arguments, ...) {
2     if (φ) {
3         //update fluents using successor state axioms
4     } else {
5         fail; //backtrack
6     }
7 }
```

where ϕ is the disjunction of one or more Boolean formulae (as specified in user-defined *possible* statements), or true if no precondition is provided.

We now observe that the body of the *if* statement is only executed if ϕ holds. Further we claim via Assumption 2 that this precondition is a correct implementation of a user-provided *possible* formula which is defined to be semantically equivalent to $Poss(\alpha[s], s)$ in the Situation Calculus. That is, a successful transition implies

$$Trans_{\text{jolog}}(\alpha[s], s, \delta', s') \Rightarrow Poss(\alpha[s], s)$$

Next we examine the body of the *if* statement itself. An assertion within the compiler ensures that the body can only contain assignments to fluents, as indicated in the above template. Furthermore these assignments correspond to user-provided *set* statements – which are defined to be semantically equivalent to the current action's successor-state axioms in the Situation Calculus. Thus by executing all of these assignments we are modifying the situation by precisely the

successor-state axioms for this action. That is, these assignments transition the current situation s to a new situation $s' = do(\alpha, s)$. We reinforce this assertion with the following argument:

Proof: Assume that executing these assignments in situation s results in a situation s'' where $s'' \neq s' = do(\alpha[s], s)$. This means that there must be at least one fluent f where its value differs between the two successor situations:

$$\exists f \mid f[s'] \neq f[s'']$$

Since both s' and s'' originated from the same previous situation s we know that either:

- $f[s] = f[s'] \neq f[s'']$. That is, the fluent value changed when it shouldn't have. Fluents, however, will only be modified if an assignment statement is compiled from a user-provided *sets* statement. Further, this *sets* statement *defines* the successor-state axiom for this domain, so $f[s']$ should *also* change. Contradiction.
- $f[s] = f[s''] \neq f[s']$. That is, the fluent value did not change when it should have. We note that for the value to remain unchanged there must have been no corresponding *sets* statement for this fluent, and so the successor-state axiom (complemented by the closed-world assumption) should ensure that the fluent value does not change in the Situation Calculus, that is $f[s] = f[s'] = f[s'']$. Contradiction.
- $f[s] \neq f[s'] \neq f[s'']$ That is, *Jolog* and the Situation Calculus both update the fluent, but get different results. Clearly they are both executing the same successor-state axiom as this is provided by the user. The compilation process must then be responsible for distorting the effects of this formula, however this contradicts Assumption 3.

So by contradiction, s'' cannot exist and we can conclude:

$$Trans_{\text{jolog}}(\alpha, s, \delta', s') \Rightarrow s' = do(\alpha, s)$$

Finally, if the JVM has completed this action – that is $Trans_{\text{jolog}}(\alpha[s], s, \delta', s')$ holds – then there is nothing left to do. Hence this subprogram must transition to the empty program $\delta' = nil$.

The conjunction of the above claims allows us to conclude that

$$Trans_{\text{jolog}}(\alpha, s, \delta', s') \Rightarrow Poss(\alpha[s], s) \wedge \delta' = nil \wedge s' = do(\alpha[s], s)$$

We prove the reverse direction of the equivalence via its contrapositive.

The Situation Calculus clearly indicates that a situation is only modified by executing an action[3] $s' = do(\alpha[s], s)$. Let us assume that this action α is currently possible and we have finished executing our immediate subprogram:

$$Poss(\alpha, s) \wedge \delta' = nil$$

Now if we were in situation s and we transitioned to a new situation $s' \neq do(\alpha[s], s)$ then we cannot have executed α to do so (because as we saw earlier we *would* be in $s' = do(\alpha[s], s)$). Accordingly, if we do not change to the situation $s' = do(\alpha[s], s)$ then we have not executed α and so $Trans_{\text{jolog}}(\alpha, s, \delta', s')$ clearly does not hold. We further justify this conclusion by appealing to the above proof of correctness of the successor-state axioms. Finally, we can combine these facts to see:

$$\begin{aligned}
\neg Trans_{\text{jolog}}(\alpha, s, \delta', s') &\Rightarrow s' \neq do(\alpha[s], s) \\
&\Rightarrow \neg Poss(\alpha[s], s) \vee \delta' \neq nil \vee s' \neq do(\alpha[s], s) \qquad (\dagger) \\
&\Rightarrow \neg \left(Poss(\alpha[s], s) \wedge \delta' = nil \wedge s' = do(\alpha[s], s) \right) \\
\therefore Trans_{\text{jolog}}(\alpha, s, \delta', s') &\Leftarrow Poss(\alpha[s], s) \wedge \delta' = nil \wedge s' = do(\alpha[s], s)
\end{aligned}$$

We draw attention to † – we assumed that $Poss$ and δ' had the *opposite* values, so they are both false in the current context. Thus the truth result here is still equivalent to the previous line. Then when we extract the negation we end up with a conjunction that still holds because of our original assumption.

Note that our assumptions that $Poss(\alpha, s, \delta', s')$ and $\delta' = nil$ holds are not strictly necessary – this proof hinges on the correctness of the situation and so arbitrary truth values for $Poss$ and δ' could technically be inserted via 'disjunctive introduction'. However this is a controversial argument form and so we avoid it at the cost of expanding our proof in the cases where these assumptions do not hold. We can see from the code template that the successor state axioms are only evaluated if the precondition holds; if $\neg Poss(\alpha, s)$ then this if statement will fail and so will the entire action – we cannot have transitioned. Alternatively, if our final state $\delta' \neq nil$

[3]This is especially true in *Reiter*'s formulation, since the situation is nothing more than the list of executed actions! Alternatively we can observe that only the *Congolog trans* definition for actions makes any claim about modification. The only other rule to even mention the situation is the test relation $?(\phi)$ which asserts that the situation does *not* change.

then we have not completed transitioning and so trans cannot hold in this case either. Hence our equivalence result above remains valid.

Finally we cite Assumption 2 above to ensure that if an action is called then a definition for that action exists. Given this we can easily claim that a primitive action is never final: it will always generate code and so the JVM will always have something to do.

Thus $Final_{\text{jolog}}(\alpha, s) \equiv False$.

7.3.3 Test

The precise nature of the *test* construct has evolved with the Golog family. In the first *Golog*, *test* behaved essentially like an assert: if we execute the *test* statement and continue onto code immediately after it, then we know that the condition is true. Otherwise execution backtracks to a previous choicepoint.

Test was redefined slightly for *Congolog* though; given sufficient concurrent priorities the test statement may *block waiting* for the condition to be made true by another 'thread' instead of simply failing. However *test* will also decay to a simple assert – as in *Golog* – in the absence of concurrent constructs[2][4].

Jolog prefers to delegate all forms of concurrency to *Java* constructs – blocking in particular should be achieved via a proper semaphore from the *Java* concurrency library – hence the implementation of *test* is done as described in the former definition, specifically:

Listing 7.2: code template for the *test* construct

```
1 if (¬φ) {
2     throw new FailedPreconditionException(); //backtrack
3 }
```

This code template indicates two potential executions:

- ϕ is true, so the *if* fails and execution of this program terminates
- ϕ is false, so the *if* succeeds and proceeds to throw an exception (and hence backtrack).

[4]Note that despite the assertion, we cannot guarantee the condition is *still* true immediately afterwards due to potential intervention by exogenous actions.

We can then observe that *successfully* running this program (that is the JVM transitions through it) yeilds several postconditions:

- ϕ is true
- the subprogram is empty (by definition), so $\delta' = nil$
- the situation has not changed, so $s = s'$

Hence:

$$Trans_{\text{jolog}}(?(\phi), s, \delta', s') \Rightarrow \phi[s] \wedge \delta' = nil \wedge s = s'$$

We now consider the reverse direction. If the situation has not changed then we may have executed a *test*, since *test* cannot be responsible for modifying the situation by Assumption 4 and because no assignments appear in this code template.

$$Trans_{\text{jolog}}(?(\phi), s, \delta', s') \Leftarrow s = s'$$

Similarly, if we have successfully transitioned to the empty program then we must have done so by executing the *test*, so:

$$Trans_{\text{jolog}}(?(\phi), s, \delta', s') \Leftarrow \delta' = nil$$

Finally, as we noticed earlier, if the precondition holds then the entire *test* since no other code in the template is able to fail:

$$Trans_{\text{jolog}}(?(\phi), s, \delta', s') \Leftarrow \phi[s]$$

Thus:

$$Trans_{\text{jolog}}(?(\phi), s, \delta', s') \Leftarrow \phi[s] \wedge \delta' = nil \wedge s = s'$$

And so *Jolog*'s *test* is equivalent to *Congolog*'s:

$$Trans_{\text{jolog}}(?(\phi), s, \delta', s') \equiv \phi[s] \wedge \delta' = nil \wedge s = s'$$

Note also that the there is no reason not to evaluate the condition, so the program cannot be considered complete until this has been performed:

$$Final_{\text{jolog}}(?(\phi), s) \equiv False$$

7.3.4 Sequences

Sequences are trivially implemented in *Java*; emitting one subprogram followed by another subprogram means the JVM will execute the first, then the second, resulting in a code template resembling:

$$\delta_1; \delta_2$$

In terms of possible interleavings we end up with two possible cases:

- Interruption at some point γ during (or immediately after) the first subprogram, in which case we transition to a new subprogram $\delta' = (\gamma; \delta_2)$: the sequence composed of the remainder of δ_1 – that is, γ – followed by δ_2.
- We finished δ_1 (so $Final_{\text{jolog}}(\delta_1, s)$ holds) and then define the transition entirely in terms of the subprogram δ_2.

$$\begin{aligned} Trans_{\text{jolog}}(\delta_1; \delta_2, s, \delta', s') \Rightarrow \exists \gamma \mid\ & (\delta' = (\gamma; \delta_2) \wedge Trans_{\text{jolog}}(\delta_1, s, \gamma, s')) \vee \\ & (Final_{\text{jolog}}(\delta_1, s) \wedge Trans_{\text{jolog}}(\delta_2, s, \delta', s')) \end{aligned}$$

For the opposite direction assume we have program $\delta_1 = \beta; \gamma$. If we have successfully emitted the subprogram β in the past (that is δ_1 successfully transitioned as far as γ), then we are at a state $\delta' = (\gamma; \delta_2)$ now.

$$Trans_{\text{jolog}}(\delta_1; \delta_2, s, \delta', s') \Leftarrow \exists \gamma \mid\ (\delta' = (\gamma; \delta_2) \wedge Trans_{\text{jolog}}(\delta_1, s, \gamma, s'))$$

Alternatively, if we have finished executing δ_1 (so it is 'final') and we successfully execute δ_2 then we will transition to some (possibly empty) subprogram δ' from here:

$$Trans_{\text{jolog}}(\delta_1; \delta_2, s, \delta', s') \Leftarrow Final_{\text{jolog}}(\delta_1, s) \wedge Trans_{\text{jolog}}(\delta_2, s, \delta', s')$$

The conjunction of these two assertions satisfies our equivalence:

$$\begin{aligned} Trans_{\text{jolog}}(\delta_1; \delta_2, s, \delta', s') \equiv \exists \gamma \mid\ & (\delta' = (\gamma; \delta_2) \wedge Trans_{\text{jolog}}(\delta_1, s, \gamma, s')) \vee \\ & (Final_{\text{jolog}}(\delta_1, s) \wedge Trans_{\text{jolog}}(\delta_2, s, \delta', s')) \end{aligned}$$

Similarly, a sequence is only 'final' if all subprograms in the sequence are also final (that is, it is reduced to a sequence of empty programs). Hence:

$$Final_{\text{jolog}}(\delta_1; \delta_2, s) \Rightarrow Final_{\text{jolog}}(\delta_1, s) \wedge Final_{\text{jolog}}(\delta_2, s)$$

Or in the opposite direction, if two consecutive subprograms δ_1 and δ_2 are final, then their conjunction clearly is too:

$$Final_{\text{jolog}}(\delta_1; \delta_2, s) \Leftarrow Final_{\text{jolog}}(\delta_1, s) \wedge Final_{\text{jolog}}(\delta_2, s)$$

7.3.5 Nondeterministic Branch (ndet)

The implementation of *ndet* in *Congolog* is trivially simple given the sophistication of the construct – either the first program works or the second does. While the same elegance is not possible with *Java* bytecode, we can see that at a higher 'meta' level we achieve the same thing. Observe in the following pseudocode that our implementation of *ndet* will run δ_1 and then break out of the *ndet* statement (with the trailing *goto*). However if δ_1 should fail then an explicit exception handler transfers control to δ_2. Further, another exception handler will catch any failure with δ_2, so if this also fails then the entire *ndet* will fail. Thus *ndet* will only transition if at least one of its subprograms can transition. Formalising this notion may be more illustrative, so we do this now.

Proof: We prove *Jolog*'s *ndet* implementation implies the *Congolog* definition by cases:

- *Case 1 – δ_1 succeeds:* then *ndet* will execute δ_1 followed by a *goto* and successfully terminate – $Trans_{\text{jolog}}$ holds.
- *Case 2 – δ_1 fails:* then δ_1 will fail (throw an exception) which will be caught by the first exception handler and control is transferred to δ_2. There are now two additional subcases to consider:
  - *Case 2a – δ_2 succeeds:* then control will proceed to the trailing *goto* and the *ndet* will still succeed – $Trans_{\text{jolog}}$ holds.
  - *Case 2b – δ_2 fails:* then the second exception handler will transfer control to the $failPoint$ and so the entire *ndet* will fail – $Trans_{\text{jolog}}$ does not hold.

Thus $Trans_{\text{jolog}}(ndet(\delta_1|\delta_2), s, \delta', s')$ will succeed if and only if either δ_1 or δ_2 succeed.

$$Trans_{\text{jolog}}(ndet(\delta_1|\delta_2), s, \delta', s') \equiv Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s')$$

We can make a similar argument about the completion of an *ndet* statement too: it will be final if either of the two subprograms is final (which ever program the *ndet* 'chooses' and successfully runs):

$$Final(ndet(\delta_1|\delta_2), s) \equiv Final(\delta_1, s) \vee Final(\delta_2, s)$$

Listing 7.3: simplified code template for nondeterministic branch (*ndet*)

```
1  doNdet1:
2      δ1
3      goto endNdet
4
5  doNdet2:
6      δ2
7      goto endNdet
8
9  failPoint:
10     throw exception //backtrack beyond the ndet statement
11
12 endNdet:
```

with the corresponding exception table:

from	to	with
doNdet1	doNdet2	doNdet2
doNdet2	handler	failPoint
endNdet	endScope	handler

In the opposite direction we note that if δ_1 can transition then the code template above will execute δ_1 then a *goto* (which does not fail). That is, the only thing preventing δ_1 (and so the *ndet* too) from executing successfully in the above *ndet* code is δ_1 itself, so:

$$Trans_{\text{jolog}}(ndet(\delta_1|\delta_2), s, \delta', s') \Leftarrow Trans(\delta_1, s, \delta', s')$$

Otherwise δ_1 fails, which in *Jolog* means it throws an exception. There is clearly an exception handler in this case that will transfer control directly to δ_2. If this program can run successfully then it will execute and then proceed to its own *goto* statement – *ndet* works once again:

$$Trans_{\text{jolog}}(ndet(\delta_1|\delta_2), s, \delta', s') \Leftarrow \neg Trans(\delta_1, s, \delta', s') \wedge Trans(\delta_2, s, \delta', s')$$

Finally we again note that if δ_2 also fails, then control is handed to the $fail$ point, which leads the entire *ndet* to fail. Thus we can conclude that the two cases are the only ones capable of leading the *ndet* to succeed. The conjunction of these two claims provides our equivalence:

$$Trans_{\text{jolog}}(ndet(\delta_1|\delta_2), s, \delta', s') \Leftarrow Trans(\delta_1, s, \delta', s') \vee (\neg Trans(\delta_1, s, \delta', s') \wedge Trans(\delta_2, s, \delta', s'))$$

Once again we assert that the same argument applies to whether *ndet* is Final and so we have equivalence here too.

7.3.6 Nondeterministic Choice of Arguments (pick)

The code for *pick* behaves in much the same way as for *ndet* – particularly in regard to saving and restoring the initial state, running the inner program and then storing a dummy state over the top for maintaining the backtrack id. The major distinction between these constructs is that *pick* has a constant code size – it sends the backtrack id directly to the fluent store which returns the next binding through internal phenomena. *Ndet* on the other hand uses the backtrack id locally, so code size grows linearly with the number of alternative programs.

Listing 7.4: simplified code template for the *pick* construct

```
1 pick:
2     x = jolog_fluent_store.getNextBinding(type(v), backtrackId)
3 user:
4     δ_x^v
5 end:
```

which is complemented with a similar exception table:

from	to	with
user	endScope	pick

Note that all notion of state has been omitted from this code template; the details of a working implementation mean this template becomes far more convoluted without affecting its applicability to the $Trans$ and $Final$ relations.

We now assert that the *getNextBinding* method contains boilerplate *Java* 'Reflection' code for iterating through the values in the fluent store and potentially all constants for enumerable types too. Given this, we assume that all potential variable bindings for variable v of type $type(v)$ will be returned if we repeatedly call this function whilst enumerating the backtrackId. Note that calling this with a backtrackId higher than the number of bindings will cause the fluent store to throw an exception, forcing the *pick* statement to backtrack.

With this in mind we can observe that the *pick* code gets the first binding and attempts to run the user program with this value. Note that v is just a local variable in our case, so δ_x^v is the same as any program δ except that in this case we expect (though do not require) it to somehow use the binding x for variable v. If this succeeds then ,smgdl

Expressed more formally we can say: Attempting to execute $Trans(pick(v)\{\delta\}, s, \delta', s')$ will lead the system to enumerate possible bindings until either one works (in which case we have found a binding $\exists x$ such that the subprogram δ transitions) or we run out of bindings. The latter case will result in an unhandled exception thrown by the $getNextBinding$ function, that is *pick* fails (it does not transition). The former case produces the following result:

$$Trans_{\text{jolog}}(pick(v)\{\delta\}, s, \delta', s') \Rightarrow \exists x \,|\, Trans_{\text{jolog}}(\delta_x^v, s, \delta', s')$$

Similarly, *pick* depends solely on the user program to become final; if we know δ can transition then we can also conclude that the form of the above equation holds for whether *pick* has completed:

$$Final_{\text{jolog}}(pick(v)\{\delta\}, s) \Rightarrow \exists x \,|\, Final_{\text{jolog}}(\delta_x^v, s)$$

Alternatively, assume that the fluent store holds some value x such that the user program δ_x^v succeeds ($Trans(\delta_x^v, s, \delta', s')$ holds). We know that *pick* will enumerate all values in the fluent store, so by our assumption the enumeration will eventually find it. Thus $Trans$ will succeed:

$$Trans_{\text{jolog}}(pick(v)\{\delta\}, s, \delta', s') \Leftarrow \exists x \,|\, Trans_{\text{jolog}}(\delta_x^v, s, \delta', s')$$

Finally, an equivalent argument to the one above holds for whether *pick* has completed:

$$Final_{\text{jolog}}(pick(v)\{\delta\}, s) \Leftarrow \exists x \,|\, Final_{\text{jolog}}(\delta_x^v, s)$$

Thus our *pick* is equivalent to *Congolog*'s definition:

$$Trans_{\text{jolog}}(pick(v)\{\delta\}, s, \delta', s') \equiv \exists x \,|\, Trans_{\text{jolog}}(\delta_x^v, s, \delta', s')$$

$$Final_{\text{jolog}}(pick(v)\{\delta\}, s) \equiv \exists x \,|\, Final_{\text{jolog}}(\delta_x^v, s)$$

7.3.7 Nondeterministic Iteration (Kleene star)

We first examine the following code template:

Listing 7.5: code template for nondeterministic iteration (*Kleene star*)

```
1  //try zero executions first ...
2  goto endKleene
3
4  //do the program once more ...
5  startKleene:
6  δ //run the user program
7
8  endKleene:
9
10 catch exception from endKleene to endScope with startKleene
```

Upon first encountering this code, the JVM will jump to the end and trivially complete the subprogram. We can consider this the base case of an induction:

$$Trans(\delta^*, s, \delta', s') \Rightarrow \gamma = \delta \wedge (\delta' = \gamma; \delta^*) \wedge Trans(\delta, s, \delta, s')$$

That is, we need not transition the user program δ at all, and the overall system transitions to a state where its remaining code is the user program δ followed by the original Kleene program. If this code is rerun then it now needs to execute δ *one* or more times[5]. This is perplexing at first; we can think of this as saying that the first 'execution' does not run δ at all (δ transitions to itself), but that the *program* transitions to a new program where zero-executions are no longer permitted. This indicates that we have already tried this case. Under this interpretation we do not run δ' just because it is the transitioned program – indeed this does not make sense since because our system has finished transitioning and the *Java* code does not mutate or change in any way after compilation. We simply note that we need to run δ one extra time if we backtrack to this point.

Now, if a subsequent error forces backtracking then it will be caught 'inside' the Kleene loop (at *startKleene*) which proceeds to run the user program and then continue. Future errors are handled in the same way: backtracking to this point and running the user program one more time, so:

$$Trans(\delta^*, s, \delta', s') \Rightarrow \exists \gamma \,|\, (\delta' = \gamma; \delta^*) \wedge Trans(\delta, s, \gamma, s')$$

[5]In common regular expression notation our program has transitioned from δ^* to $\delta+$.

where γ may be *nil* to indicate that we have finished another execution of δ and that we can execute zero or more in the future after backtracking to this point.

For the reverse direction we can instead consider γ to be some arbitrary length sequence of δ programs. If we have already transitioned from a single δ to a sequence of them – as the *Kleene* construct does – then we must have rerun the program several times (the length of the sequence many). Thus if we have transitioned to γ then this is the same as transitioning from δ^* to our exact sequence γ. However the construct is happy to execute even further in the future, so we append δ^* to the transitioned program δ' to indicate this.

$$Trans(\delta^*, s, \delta', s') \Leftarrow \exists \gamma \,|\, (\delta' = \gamma; \delta^*) \wedge Trans(\delta, s, \gamma, s')$$

We can also see that this implementation allows an arbitrary number of executions of δ – including zero – so this construct is trivially final.

$$Final(\delta^*, s) \equiv True$$

As a final note, observe that only one exception handler (the mechanism for backtracking) occurs in this code template and that serves to catch 'future' exceptions. This means that an exception within the user program δ will not be caught by this operator and so nondeterministic iteration *can* fail. These semantics not only reduce code size, but also turn out to be a useful feature of this operator, particularly when used in the implementation of while loops.

7.3.8 Function Calls

The implementation of function calls has been introduced earlier, however, here we are concerned with the *idea* of a 'function call' as it relates to first-order logic (the Situation Calculus). Previous Gologs have done this as macro expansion, we argue that essentially the same process is being performed by the JVM. Let us consider some function $\rho(\vec{x})$ defined by the predicate:

$$proc(\rho(\vec{x}), \delta)$$

When invoking a function the JVM jumps to the function's enclosed code – its definition – and runs this code too. The function call itself is thus irrelevant to the Situation Calculus; it is simply a mechanism for breaking up code. Note this approach allows recursive functions too. We now suggest that transitioning on the function call is equivalent to transitioning on the function

definition:

$$Trans_{\text{jolog}}(\rho(\vec{x}), s, \delta', s') \equiv proc(\rho(\vec{x}), \delta) \wedge Trans_{\text{jolog}}(\delta, s, \delta', s')$$

Similarly, if we are replacing execution of the function call with execution of the function body then the same claim holds for whether a program is final:

$$Final_{\text{jolog}}(\rho(\vec{x}), s, \delta', s') \equiv proc(\rho(\vec{x}), \delta) \wedge Final_{\text{jolog}}(\delta, s, \delta', s')$$

Finally, we observe that local variables are not for free – we need to distinguish them for the sake of the Situation Calculus (although this is clearly given to us within the JVM). We discharge this proof obligation with naming rules that effectively map local variables within a scope to unique names. We know that variable names are unique within their scope so it suffices for a variable with local name v, of type t and at scope depth d to have a corresponding name in the Situation Calculus composed of these components:

$$v\_t\_d$$

Note that there are many solutions to this problem, however, this should be sufficient indication that the problem is not particularly hard or important.

7.3.9 Others

It may have been noted that a few reasonably important constructs are missing from the above discussion: *if*, *while* and *for* statements. Due to the complications of nondeterministic execution we adopt the *Congolog* approach and compile these in terms of the *ndet* and *Kleene* operators. This has two major benefits:

- *ndet* and *Kleene* have more intuitive semantics in the presence of backtracking; what does it mean to backtrack into a while loop or an if statement anyway?
- it discharges proof obligations – we know *ndet* and *Kleene* work and we know sequence can compose subprograms. Why not exploit this?

These advantages are not for free. The code for *if*, *while* and *for* statements is longer and more complicated now, and worse, involves duplicate evaluation of the condition (particularly so that *ndet* doesn't try to execute the alternative just because the *if* failed). These failings will generally only be evident in conditions that involve function calls. However these functions threaten to backtrack too, so they really should be reevaluated anyway. Note also that this means that *if*, *while* and *for* statements implemented this way cannot appear in static functions – there is no backtrack stack for them to rely on. Accordingly, these constructs use the original definitions – more intuitive and condensed code – when used in a static context. Note that static functions are assumed to be for setup and handler code (like the *main* function) since they do not have access to fluents or nondeterminism, so we omit their proof obligations as being irrelevant to the Situation Calculus.

Chapter 8

Conclusions

This thesis has been a great opportunity to complement my undergraduate studies with yet unexplored fields, particularly with concepts of compilers. It has also provided me with an excuse to work on a large project incorporating both C++ and Java development and to research and implement new language constructs for the Java Virtual Machine. Breaking Prolog's monopoly on some amazing features like nondeterministic variable instantiation and backtracking into functions that have terminated – or even Python's tuples – has been a turbulent but rewarding experience. I have also developed a much greater understanding of – and appreciation for – the design and flexibility of *Java* as a language and the overall operation of the JVM.

At this point we reflect on what has been achieved:

- A compiler capable of compiling a significant subset of the *Java* grammar into *Java* bytecode (with assistance from *Jasmin*).
- The successful introduction of nondeterminism to the JVM.
- An argument for equivalence between our new language *Jolog* and the Situation Calculus.
- The successful introduction of types and type checking to our new Golog variant.
- The enhancement of existing communication protocols and the introduction of new primitives to encourage their use.
- A more familiar syntax – including sugar for tuples and I/O operations – and better syntactic checks during compilation.
- Documentation of some of the more important features of *Java*, *Jasmin* and *Boost::spirit*.

We observe that this perfectly complements our initial aims.

8.1 Future Work

Jolog can do a significant subset of things that one might want from a cognitive robotics language. However there are still many things that would make welcome extensions:

- Introducing Prolog-style pattern matching and variable substitution for function calls.
- Expanding the object-oriented nature of the language, including resolving the unclear dynamics of static fluents.
- Enhanced type-checking, especially by enforcing tuple types and introducing type hierarchies to eliminate spurious typecasts.
- Theorem proving or other mathematical conclusions in regard to the behaviour of *Jolog* programs.
- Replace the Fluent resolution mechanism with 'fluent templates'. That is, dynamic fluents are still allowed, but their name (and possibly their type) must conform to some template definition. This would make name resolution much stricter, typing much easier (and safer) and free up the compiler to happily use predicates as fluent names and values.
- Developing decision-theoretic, stochastic functions and search features available from previous Gologs, particularly *Readylog*.

Appendix A

Jolog Grammars

Listing A.1: JologGrammar.h

```
1  /*
2   * JologGrammar.h
3   *
4   *  Created on: 2/08/2009
5   *      Author: timothyc
6   */
7
8  #ifndef JOLOG_GRAMMAR
9  #define JOLOG_GRAMMAR
10
11 #include <string>
12 #include <sstream>
13 #include <cstring>
14
15 #include "GrammarUtility.h"
16 #include "LiteralGrammar.h"
17 #include "IdentifierGrammar.h"
18 #include "TypeNameGrammar.h"
19 #include "ExpressionGrammar.h"
20 #include "StatementGrammar.h"
21 #include "SkipGrammar.h"
22
```

```
23 using std::string;
24 using namespace boost::spirit::classic;
25
26 class JologGrammar : public grammar<JologGrammar> {
27    public:
28       typedef position_iterator<const char*> iterator_t;
29       typedef node_iter_data_factory<int> factory_t;
30
31       template <typename ScannerT>
32       class definition {
33          public:
34             typedef ScannerT scanner_t;
35             definition(JologGrammar const& self) :
36                expression(self.getErrorReporter()), statement(self.getErrorRepo
37
38                jolog_file
39                = !jolog_package
40                >> *jolog_import
41                >> !jolog_agent
42                >> (end_p | (lexeme_d[ +anychar_p ][self.report(file_tag)] >> no
43                ;
44
45                jolog_package
46                = root_node_d[ str_p("package") ]
47                               >> (lexeme_d[ list_p(
48                                     token_node_d[ +(alnum_p | '_') ],
49                                     no_node_d[ ch_p('.') ]
50                               )]
51                               | error[self.report(package_tag)]
52                               )
53                               >> (no_node_d[ ch_p(';') ] | error[self.report(ex
54                               ;
55
56                jolog_import
```

```
57              = root_node_d[ str_p("import") ]
58                             >> (lexeme_d[ list_p(
59                                    token_node_d[ +(alnum_p | '_' | '*') ],
60                                    no_node_d[ ch_p('.') ] )
61                                    ]
62                                    | error[self.report(import_tag)]
63                             )
64                             >> (no_node_d[ ch_p(';') ] | error[self.report(ex
65                             ;
66
67              jolog_agent
68              = *prefix >> str_p("agent")
69              >> (typeName | error[self.report(type_tag)])
70              >> ( *( implements | extends )
71                    >> no_node_d[ ch_p('{') ] //| (error[self.report(class_tag
72                                  >> *(jolog_agent_contents) // | ((+(error -
73                                  >> no_node_d[ ch_p('}') ])
74                                  | error[self.report(agent_tag)]
75                                          ;
76
77              implements
78              = root_node_d[ str_p("implements") ] >> (typeName | error[self.r
79
80              extends
81              = root_node_d[ str_p("extends") ] >> (typeName | error[self.repo
82
83              jolog_agent_contents
84              = action_definition
85              | function_definition
86              | fluent_definition
87              | lexeme_d[ +(anychar_p - '}' - eol_p) ][self.report(agent_conte
88                                                      ;
89
90              action_definition
```

```
91              = *prefix
92              >> (str_p("prim_action") | "exog_action" | "sensing_action")
93              //we either have a pattern (for exog_actions) or a function: nam
94              >> (((regex_pattern
95                    | (identifier
96                          >> no_node_d[ ch_p('(') ]
97                                       >> !list_p(parameter_definition, no_no
98                                       >> no_node_d[ ch_p(')') ])
99              )
100             //                        >> ((no_node_d[ ch_p('(') ]  >> !list_p
101             >> no_node_d[ ch_p('{') ]
102                           >> *( ("possible"
103                                 >> (expression | error[self.report(action_de
104                                 >> (no_node_d[ ch_p(';') ] | error[self.repo
105                           )
106
107                           | ("set"
108                                 >> (expression | error[self.report(action_de
109                                 //>> (no_node_d[ ch_p('=') ] | error[self.re
110                                 //>> (expression | error[self.report(action_
111                                 >> !(str_p("if") >> expression)
112                                 >> (no_node_d[ ch_p(';') ] | error[self.repo
113                           )
114
115                           | ("send"
116                                 >> (statement | error[self.report(action_def
117                           )
118
119                           | ("return"
120                                 >> (expression | error[self.report(action_de
121                                 >> (no_node_d[ ch_p(';') ] | error[self.repo
122                           )
123                           )
124                          >> (no_node_d[ ch_p('}') ] | error[self.report(act
```

```
125                 )
126                 | error[self.report(action_definition_tag)]
127                 )
128                 ;
129
130                 fluent_definition
131                 = *prefix >> typeName
132                 >> identifier
133                 >> !(no_node_d[ ch_p('(') ]
134                                 >> !list_p(expression, no_node_d[ ch_p(',')])
135                                 >> no_node_d[ ch_p(')') ]
136                 ) //| (error[self.variable_error] /*>> nothing_p*/))
137                 >> !('=' >> (initialiser | (error[self.report(expression_tag)] /
138                 >> (no_node_d[ ch_p(';') ] | eps_p[self.report(expression_statem
139                 ;
140
141                 function_definition
142                 = !privacy >> *prefix >> typeName >> identifier
143                 >> ((no_node_d[ ch_p('(') ] >> !list_p(parameter_definition, no
144                 >> no_node_d[ ch_p('{') ]
145                 >> *(statement)
146                 >> (no_node_d[ ch_p('}') ] | error[self.report(scope_tag)])
147                 ;
148
149                 parameter_definition
150                 = *(str_p("transient")) >> typeName >> identifier;
151
152                 prefix
153                 = str_p("final") | "static" | "abstract";
154
155                 initialiser
156                 = (root_node_d[ ch_p('{') ] >> infix_node_d[ !list_p(initialiser
157                 | expression; //zero allowed?
158
```

```
159             privacy
160             = str_p("public") | "private" | "protected";
161
162             regex_pattern //TODO do we want to use escapes? to allow " for e
163             = no_node_d[ ch_p('"') ] >> token_node_d[ lexeme_d[ *(anychar_p -
164
165             error
166             //= lexeme_d[ +anychar_p ]
167             //= (anychar_p - space_p)
168             //= token_node_d[ ( +(alnum_p | '_') | ((ch_p('[') | '{' | '(') )
169             = token_node_d[ +(alnum_p | space_p | '_')
170                             | ((!(ch_p('[') | '{' | '(')) >> *(anychar_p - (
171                           ]
172             ;
173         }
174
175         rule<scanner_t, parser_tag<file_tag> > const& start() const {
176             return jolog_file;
177         }
178
179     private:
180         LiteralGrammar literal;
181         IdentifierGrammar identifier;
182         ExpressionGrammar expression;
183         StatementGrammar statement;
184         TypeNameGrammar typeName;
185
186         rule<scanner_t, parser_tag<file_tag> > jolog_file;
187         rule<scanner_t, parser_tag<package_tag> > jolog_package;
188         rule<scanner_t, parser_tag<import_tag> > jolog_import;
189         rule<scanner_t, parser_tag<agent_tag> > jolog_agent;
190         rule<scanner_t, parser_tag<implements_tag> > implements;
191         rule<scanner_t, parser_tag<extends_tag> > extends;
192         rule<scanner_t, parser_tag<agent_contents_tag> > jolog_agent_conten
```

```
193             rule<scanner_t, parser_tag<action_definition_tag> > action_definitio
194             rule<scanner_t, parser_tag<fluent_definition_tag> > fluent_definitio
195             rule<scanner_t, parser_tag<function_definition_tag> > function_defi
196             rule<scanner_t, parser_tag<parameter_definition_tag> > parameter_de
197             rule<scanner_t, parser_tag<privacy_tag> > privacy;
198             rule<scanner_t, parser_tag<prefix_tag> > prefix;
199             rule<scanner_t, parser_tag<string_tag> > regex_pattern;
200             rule<scanner_t, parser_tag<initialiser_tag> > initialiser;
201
202             rule<scanner_t, parser_tag<error_tag> > error;
203         };
204
205         JologGrammar(std::ostream& stderr, bool& errorFlag) :
206             stderr(stderr), errorFlag(errorFlag) {}
207
208         ErrorMessage getErrorReporter() const {
209             return ErrorMessage(file_tag, stderr, errorFlag);
210         }
211
212         ErrorMessage report(ParserTags tag) const {
213             return ErrorMessage(tag, stderr, errorFlag);
214         }
215
216         tree_parse_info<iterator_t, factory_t> parseString(const char* s, string
217             iterator_t begin(s, s+strlen(s), filename);
218             iterator_t end;
219             SkipGrammar skip;
220             return ast_parse<factory_t>(begin, end, *this, skip);
221         }
222
223     private:
224         std::ostream& stderr;
225         bool& errorFlag;
226 };
```

```
227
228 #endif
```

Listing A.2: StatementGrammar.h

```
1  /*
2   * StatementGrammar.h
3   *
4   *  Created on: 2/08/2009
5   *      Author: timothyc
6   */
7
8  #ifndef STATEMENT_GRAMMAR
9  #define STATEMENT_GRAMMAR
10
11 #include <string>
12 #include <sstream>
13
14 #include "GrammarUtility.h"
15 #include "ExpressionGrammar.h"
16 #include "IdentifierGrammar.h"
17
18 using std::string;
19 using namespace boost::spirit::classic;
20
21 class StatementGrammar : public grammar<StatementGrammar> {
22    private:
23       ErrorMessage errorReporter;
24
25    public:
26       typedef position_iterator<const char*> iterator_t;
27       typedef node_iter_data_factory<int> factory_t;
28
29       StatementGrammar(ErrorMessage e) : errorReporter(e) {}
30
31       ErrorMessage report(ParserTags tag) const {
```

```
32          return errorReporter.report(tag);
33      }
34
35      template <typename ScannerT>
36      class definition {
37          public:
38              typedef ScannerT scanner_t;
39              definition(StatementGrammar const& self) : expression(self.errorRep
40
41                  scope //may infinite loop ... remove nothing_p
42                      = root_node_d[ ch_p('{') ]
43                      >> *(statement) // /*| ((error - '}')[self.statement_error] *
44                      >> (no_node_d[ ch_p('}') ]); // | ((error - '}')[self.stateme
45
46                  statement
47                      = jump_statement
48                      | local_variable_definition
49                      | if_statement
50                      | while_statement
51                      | for_statement
52                      | holds
53                      | ndet
54                      | kleene
55                      | pick
56                      | scope
57                      | !expression >> (no_node_d[ ch_p(';') ]) // | (+alnum_p)[sel
58                      | ((+(alnum_p - '}'))[self.report(expression_statement_tag)])
59                      ;
60
61                  if_statement
62                      = root_node_d[ str_p("if") ] >> no_node_d[ ch_p('(') ] >> exp
63                      >> scope
64                      >> !else_statement;
65
```

```
66                  else_statement
67                     = no_node_d[ str_p("else") ] >> (if_statement | scope);
68
69                  while_statement
70                     = root_node_d[ str_p("while") ] >> no_node_d[ ch_p('(') ] >>
71                     >> scope
72                     ;
73
74                  for_statement
75                     = no_node_d[ str_p("for") >> '(' ]
76                     >> ((!expression >> ';' >> !expression >> ';' >> !expression)
77                     >> ch_p(')')
78                     >> scope
79                     ;
80
81                  jump_statement
82                     = root_node_d[ str_p("return") ]
83                     >> !expression
84                     >> (no_node_d[ ch_p(';') ] | ((+alnum_p)[self.report(expressio
85                     ;
86
87                  local_variable_definition
88                     = *(str_p("transient"))
89                     >> typeName
90                     >> identifier
91                     >> !( ('=' >> expression)
92                           | *(',' >> identifier)
93                     )
94                     >> no_node_d[ ch_p(';') ]
95                     ;
96
97                  ndet
98                     = root_node_d[ str_p("ndet") ]
99                     >> no_node_d[ ch_p('(') ]
```

```
100                 >> list_p(statement, no_node_d[ch_p('|')])
101                 >> no_node_d[ ch_p(')') ]
102                 ;
103
104             pick
105                 = root_node_d[str_p("pick") | "some" | "all"]
106                 >> no_node_d[ch_p('(')] >> typeName >> identifier >> no_node_
107                 >> scope
108                 ;
109
110             holds
111                 = root_node_d[ch_p('?')]
112                 >> no_node_d[ch_p('(')]
113                 >> expression
114                 >> no_node_d[ ch_p(')') >> ch_p(';') ]
115                 ;
116
117             kleene
118                 = (scope | expression)
119                 >> root_node_d[ch_p('*')]
120                 >> no_node_d[ch_p(';')]
121                 ;
122
123         }
124
125         rule<scanner_t, parser_tag<statement_tag> > const& start() const {
126             return statement;
127         }
128
129     private:
130         IdentifierGrammar identifier;
131         TypeNameGrammar typeName;
132         ExpressionGrammar expression;
133
```

```
134             rule<scanner_t, parser_tag<scope_tag> > scope;
135             rule<scanner_t, parser_tag<jump_tag> > jump_statement;
136             rule<scanner_t, parser_tag<statement_tag> > statement;
137             rule<scanner_t, parser_tag<if_tag> > if_statement;
138             rule<scanner_t, parser_tag<else_tag> > else_statement;
139             rule<scanner_t, parser_tag<while_tag> > while_statement;
140             rule<scanner_t, parser_tag<for_tag> > for_statement;
141             rule<scanner_t, parser_tag<local_tag> > local_variable_definition;
142             rule<scanner_t, parser_tag<ndet_tag> > ndet;
143             rule<scanner_t, parser_tag<kleene_tag> > kleene;
144             rule<scanner_t, parser_tag<pick_tag> > pick;
145             rule<scanner_t, parser_tag<holds_tag> > holds;
146
147         };
148
149 };
150
151 #endif
```

Listing A.3: ExpressionGrammar.h

```
1  /*
2   * ExpressionGrammar.h
3   *
4   *  Created on: 2/08/2009
5   *      Author: timothyc
6   */
7
8  #ifndef EXPRESSION_GRAMMAR
9  #define EXPRESSION_GRAMMAR
10
11 #include <string>
12 #include <sstream>
13
14 #include "GrammarUtility.h"
15 #include "LiteralGrammar.h"
```

```
16 #include "IdentifierGrammar.h"
17 #include "TypeNameGrammar.h"
18
19 using std::string;
20 using namespace boost::spirit::classic;
21
22 class ExpressionGrammar : public grammar<ExpressionGrammar> {
23    private:
24       ErrorMessage errorReporter;
25
26    public:
27       typedef position_iterator<const char*> iterator_t;
28       typedef node_iter_data_factory<int> factory_t;
29
30       ExpressionGrammar(ErrorMessage e) : errorReporter(e) {}
31
32       ErrorMessage report(ParserTags tag) const {
33          return errorReporter.report(tag);
34       }
35
36       template <typename ScannerT>
37       class definition {
38          public:
39             typedef ScannerT scanner_t;
40             definition(ExpressionGrammar const& self) {
41
42                expression
43                = list_p(assignment_expression, longest_d[ str_p("<<") | ">>" ])
44                | (token_node_d[ +(anychar_p - ';' - '}') ][self.report(expressio
45                // | eps_p[self.report(expression_tag)]
46                ;
47
48                assignment_expression
49                = conditional_expression >> !(root_node_d[ assignment_operator ]
```

```
50
51              assignment_operator
52              = longest_d[ (str_p("*=") | "/=" | "%=" | "+=" | "-=" | '=') ];
53
54              conditional_expression
55              = logical_or_expression >> !('?' >> expression >> ':' >> express
56
57              logical_or_expression
58              = list_p(logical_and_expression, root_node_d[ str_p("||") ]);
59
60              logical_and_expression
61              //= list_p(inclusive_or_expression, root_node_d[ str_p("&&") ]);
62              = list_p(relational_expression, root_node_d[ str_p("&&") ]);
63
64              relational_expression
65              = list_p(additive_expression, longest_d[ (str_p("==") | "!=" | '
66
67              additive_expression
68              = list_p(multiplicative_expression, root_node_d[ (ch_p('+') | '-
69
70              multiplicative_expression
71              = list_p(cast_expression, root_node_d[ (ch_p('*') | '/' | '%') ]
72
73              cast_expression
74              = *(inner_node_d[ '(' >> typeName >> ')' ]) >> unary_expression;
75
76              unary_expression
77              = (root_node_d[ str_p("++") | "--" ] >> unary_expression)
78              | (root_node_d[ (ch_p('+') | '-' | '!') ] >> cast_expression)
79              | postfix_expression
80              ;
81
82              /*postfix_expression
83                                              = atomic_expression
```

```
84                                                  >> *( array_reference
85                                                       | function_call
86                                                       | member_reference
87                                                       | (str_p("++") | "--")
88                                                       )
89                                                  ;*/
90
91              postfix_expression
92              = atomic_expression
93              >> *( (root_node_d[ ch_p('[') ] >> expression >> no_node_d[ ch_p
94                    | (root_node_d[ ch_p('(') ] >> !list_p(assignment_expressio
95                    | (root_node_d[ ch_p('.') ] >> postfix_expression)
96                    | (root_node_d[ str_p("++") | "--" ])
97              )
98              ;
99
100             /*                                        array_reference
101                                                 = root_node_d[ ch_p('[') ]
102                                                 >> expression
103                                                 >> no_node_d[ ch_p(']') ]
104                                                 ;
105
106                                       function_call
107                                                 = root_node_d[ ch_p('(') ]
108                                                 >> !list_p(assignment_expressio
109                                                 >> no_node_d[ ch_p(')') ]
110                                                 ;
111
112                                       member_reference
113                                                 = root_node_d[ ch_p('.') ] >> p
114
115             atomic_expression
116             = literal
117             | (root_node_d[ str_p("new") ] >> typeName >> ch_p('(') >> !list
```

```
118             | (root_node_d[ str_p("new") ] >> typeName >> ch_p('[') >> condi
119             | someAll
120             | identifier
121             | tuple
122             | (no_node_d[ ch_p('(') ] >> root_node_d[ expression ] >> no_nod
123             ;
124
125             tuple
126             = root_node_d[ ch_p('(') ]
127                           >> expression >> +(no_node_d[ ch_p(',') ] >> expr
128                           >> no_node_d[ ch_p(')') ]
129                                         ;
130
131             someAll
132             = root_node_d[ str_p("some") | "all" ]
133             >> no_node_d[ch_p('(')]
134             >> typeName
135             >> identifier
136             >> no_node_d[ch_p('|')]
137             >> expression
138             >> no_node_d[ch_p(')')]
139             ;
140
141         }
142
143         rule<scanner_t, parser_tag<expression_tag> > const& start() const {
144             return expression;
145         }
146
147     private:
148         LiteralGrammar literal;
149         IdentifierGrammar identifier;
150         TypeNameGrammar typeName;
151
```

```
152             rule<scanner_t, parser_tag<expression_tag> > expression;
153             rule<scanner_t, parser_tag<assignment_expression_tag> > assignment_
154             rule<scanner_t, parser_tag<assignment_operator_tag> > assignment_op
155             rule<scanner_t, parser_tag<conditional_expression_tag> > conditiona
156             rule<scanner_t, parser_tag<logical_or_expression_tag> > logical_or_e
157             rule<scanner_t, parser_tag<logical_and_expression_tag> > logical_an
158             rule<scanner_t, parser_tag<relational_expression_tag> > relational_e
159             rule<scanner_t, parser_tag<additive_expression_tag> > additive_expr
160             rule<scanner_t, parser_tag<multiplicative_expression_tag> > multipl
161             rule<scanner_t, parser_tag<cast_expression_tag> > cast_expression;
162             rule<scanner_t, parser_tag<unary_expression_tag> > unary_expression
163             rule<scanner_t, parser_tag<postfix_expression_tag> > postfix_expres
164             rule<scanner_t, parser_tag<array_tag> > array_reference;
165             rule<scanner_t, parser_tag<function_tag> > function_call;
166             rule<scanner_t, parser_tag<member_tag> > member_reference;
167             rule<scanner_t, parser_tag<atomic_expression_tag> > atomic_expressio
168             rule<scanner_t, parser_tag<tuple_tag> > tuple;
169             rule<scanner_t, parser_tag<pick_tag> > someAll;
170         };
171
172 };
173
174 #endif
```

Listing A.4: IdentifierGrammar.h

```
1  /*
2   * IdentifierGrammar.h
3   *
4   *  Created on: 2/08/2009
5   *      Author: timothyc
6   */
7
8  #ifndef IDENTIFIER_GRAMMAR
9  #define IDENTIFIER_GRAMMAR
10
```

```
11 #include <string>
12 #include <sstream>
13
14 #include "GrammarUtility.h"
15
16 using std::string;
17 using namespace boost::spirit::classic;
18
19 class IdentifierGrammar : public grammar<IdentifierGrammar> {
20    public:
21       typedef position_iterator<const char*> iterator_t;
22       typedef node_iter_data_factory<int> factory_t;
23
24       template <typename ScannerT>
25       class definition {
26          public:
27             typedef ScannerT scanner_t;
28             definition(IdentifierGrammar const& self) {
29                identifier
30                = no_node_d[*space_p] >> token_node_d[ lexeme_d[ ((alpha_p | '_'
31                                                                  ;
32
33             }
34
35             rule<scanner_t, parser_tag<identifier_tag> > const& start() const {
36                return identifier;
37             }
38
39          private:
40             rule<scanner_t, parser_tag<identifier_tag> > identifier;
41       };
42
43 };
44
```

```
45
46 #endif
```

Listing A.5: TypeNameGrammar.h

```
1  /*
2   * TypeNameGrammar.h
3   *
4   *  Created on: 4/08/2009
5   *      Author: timothyc
6   */
7
8  #ifndef TYPENAMEGRAMMAR
9  #define TYPENAMEGRAMMAR
10
11 #include <string>
12 #include <sstream>
13
14 #include "GrammarUtility.h"
15
16 using std::string;
17 using namespace boost::spirit::classic;
18
19 class TypeNameGrammar : public grammar<TypeNameGrammar> {
20    public:
21       typedef position_iterator<const char*> iterator_t;
22       typedef node_iter_data_factory<int> factory_t;
23
24       template <typename ScannerT>
25       class definition {
26          public:
27             typedef ScannerT scanner_t;
28             definition(TypeNameGrammar const& self) {
29                typeName
30                = lexeme_d[ token_node_d[ ((alpha_p | '_') >> *(alnum_p | '_'))
31                //>> !(no_node_d[ ch_p('<') ] >> list_p(typeName, no_node_d[ ch_
```

```
32                  >> *str_p("[]")
33                  ;
34
35              }
36
37              rule<scanner_t, parser_tag<type_tag> > const& start() const {
38                  return typeName;
39              }
40
41          private:
42              rule<scanner_t, parser_tag<type_tag> > typeName;
43      };
44
45 };
46
47 #endif
```

Listing A.6: LiteralGrammar.h

```
1  /*
2   * JologGrammar.h
3   *
4   *  Created on: 2/08/2009
5   *      Author: timothyc
6   */
7
8  #ifndef JOLOG_GRAMMAR
9  #define JOLOG_GRAMMAR
10
11 #include <string>
12 #include <sstream>
13 #include <cstring>
14
15 #include "GrammarUtility.h"
16 #include "LiteralGrammar.h"
17 #include "IdentifierGrammar.h"
```

```
18 #include "TypeNameGrammar.h"
19 #include "ExpressionGrammar.h"
20 #include "StatementGrammar.h"
21 #include "SkipGrammar.h"
22
23 using std::string;
24 using namespace boost::spirit::classic;
25
26 class JologGrammar : public grammar<JologGrammar> {
27    public:
28       typedef position_iterator<const char*> iterator_t;
29       typedef node_iter_data_factory<int> factory_t;
30
31       template <typename ScannerT>
32       class definition {
33          public:
34             typedef ScannerT scanner_t;
35             definition(JologGrammar const& self) :
36                expression(self.getErrorReporter()), statement(self.getErrorRepo
37
38                jolog_file
39                = !jolog_package
40                >> *jolog_import
41                >> !jolog_agent
42                >> (end_p | (lexeme_d[ +anychar_p ][self.report(file_tag)] >> no
43                ;
44
45                jolog_package
46                = root_node_d[ str_p("package") ]
47                               >> (lexeme_d[ list_p(
48                                     token_node_d[ +(alnum_p | '_') ],
49                                     no_node_d[ ch_p('.') ]
50                               )]
51                               | error[self.report(package_tag)]
```

```
52                          )
53                          >> (no_node_d[ ch_p(';') ] | error[self.report(ex
54                          ;
55
56                  jolog_import
57                  = root_node_d[ str_p("import") ]
58                          >> (lexeme_d[ list_p(
59                                  token_node_d[ +(alnum_p | '_' | '*') ],
60                                  no_node_d[ ch_p('.') ] )
61                                  ]
62                                  | error[self.report(import_tag)]
63                          )
64                          >> (no_node_d[ ch_p(';') ] | error[self.report(ex
65                          ;
66
67                  jolog_agent
68                  = *prefix >> str_p("agent")
69                  >> (typeName | error[self.report(type_tag)])
70                  >> ( *( implements | extends )
71                          >> no_node_d[ ch_p('{') ] //| (error[self.report(class_tag
72                                  >> *(jolog_agent_contents) // | ((+(error -
73                                  >> no_node_d[ ch_p('}') ])
74                                  | error[self.report(agent_tag)]
75                                          ;
76
77                  implements
78                  = root_node_d[ str_p("implements") ] >> (typeName | error[self.r
79
80                  extends
81                  = root_node_d[ str_p("extends") ] >> (typeName | error[self.repo
82
83                  jolog_agent_contents
84                  = action_definition
85                  | function_definition
```

```
86                  | fluent_definition
87                  | lexeme_d[ +(anychar_p - '}' - eol_p) ][self.report(agent_conte
88                                                          ;
89
90                  action_definition
91                  = *prefix
92                  >> (str_p("prim_action") | "exog_action" | "sensing_action")
93                  //we either have a pattern (for exog_actions) or a function: nam
94                  >> (((regex_pattern
95                        | (identifier
96                              >> no_node_d[ ch_p('(') ]
97                                          >> !list_p(parameter_definition, no_no
98                                          >> no_node_d[ ch_p(')') ])
99                  )
100                 //                        >> ((no_node_d[ ch_p('(') ] >> !list_p
101                 >> no_node_d[ ch_p('{') ]
102                             >> *( ("possible"
103                                    >> (expression | error[self.report(action_de
104                                    >> (no_node_d[ ch_p(';') ] | error[self.repo
105                             )
106
107                             | ("set"
108                                    >> (expression | error[self.report(action_de
109                                    //>> (no_node_d[ ch_p('=') ] | error[self.re
110                                    //>> (expression | error[self.report(action_
111                                    >> !(str_p("if") >> expression)
112                                    >> (no_node_d[ ch_p(';') ] | error[self.repo
113                             )
114
115                             | ("send"
116                                    >> (statement | error[self.report(action_def
117                             )
118
119                             | ("return"
```

```
120                             >> (expression | error[self.report(action_de
121                             >> (no_node_d[ ch_p(';') ] | error[self.repo
122                         )
123                         )
124                       >> (no_node_d[ ch_p('}') ] | error[self.report(act
125             )
126             | error[self.report(action_definition_tag)]
127             )
128             ;
129
130             fluent_definition
131             = *prefix >> typeName
132             >> identifier
133             >> !(no_node_d[ ch_p('(') ]
134                           >> !list_p(expression, no_node_d[ ch_p(',')])
135                           >> no_node_d[ ch_p(')') ]
136             ) //| (error[self.variable_error] /*>> nothing_p*/))
137             >> !('=' >> (initialiser | (error[self.report(expression_tag)] /
138             >> (no_node_d[ ch_p(';') ] | eps_p[self.report(expression_statem
139             ;
140
141             function_definition
142             = !privacy >> *prefix >> typeName >> identifier
143             >> ((no_node_d[ ch_p('(') ] >> !list_p(parameter_definition, no
144             >> no_node_d[ ch_p('{') ]
145             >> *(statement)
146             >> (no_node_d[ ch_p('}') ] | error[self.report(scope_tag)])
147             ;
148
149             parameter_definition
150             = *(str_p("transient")) >> typeName >> identifier;
151
152             prefix
153             = str_p("final") | "static" | "abstract";
```

```
154
155             initialiser
156             = (root_node_d[ ch_p('{') ] >> infix_node_d[ !list_p(initialiser
157             | expression; //zero allowed?
158
159             privacy
160             = str_p("public") | "private" | "protected";
161
162             regex_pattern //TODO do we want to use escapes? to allow " for e
163             = no_node_d[ ch_p('"') ] >> token_node_d[ lexeme_d[ *(anychar_p -
164
165             error
166             //= lexeme_d[ +anychar_p ]
167             //= (anychar_p - space_p)
168             //= token_node_d[ ( +(alnum_p | '_') | ((ch_p('[') | '{' | '(') )
169             = token_node_d[ +(alnum_p | space_p | '_')
170                             | ((!(ch_p('[') | '{' | '(')) >> *(anychar_p - (
171                           ]
172             ;
173         }
174
175         rule<scanner_t, parser_tag<file_tag> > const& start() const {
176             return jolog_file;
177         }
178
179     private:
180         LiteralGrammar literal;
181         IdentifierGrammar identifier;
182         ExpressionGrammar expression;
183         StatementGrammar statement;
184         TypeNameGrammar typeName;
185
186         rule<scanner_t, parser_tag<file_tag> > jolog_file;
187         rule<scanner_t, parser_tag<package_tag> > jolog_package;
```

```
188             rule<scanner_t, parser_tag<import_tag> > jolog_import;
189             rule<scanner_t, parser_tag<agent_tag> > jolog_agent;
190             rule<scanner_t, parser_tag<implements_tag> > implements;
191             rule<scanner_t, parser_tag<extends_tag> > extends;
192             rule<scanner_t, parser_tag<agent_contents_tag> > jolog_agent_conten
193             rule<scanner_t, parser_tag<action_definition_tag> > action_definitio
194             rule<scanner_t, parser_tag<fluent_definition_tag> > fluent_definitio
195             rule<scanner_t, parser_tag<function_definition_tag> > function_defi
196             rule<scanner_t, parser_tag<parameter_definition_tag> > parameter_de
197             rule<scanner_t, parser_tag<privacy_tag> > privacy;
198             rule<scanner_t, parser_tag<prefix_tag> > prefix;
199             rule<scanner_t, parser_tag<string_tag> > regex_pattern;
200             rule<scanner_t, parser_tag<initialiser_tag> > initialiser;
201
202             rule<scanner_t, parser_tag<error_tag> > error;
203         };
204
205         JologGrammar(std::ostream& stderr, bool& errorFlag) :
206             stderr(stderr), errorFlag(errorFlag) {}
207
208         ErrorMessage getErrorReporter() const {
209             return ErrorMessage(file_tag, stderr, errorFlag);
210         }
211
212         ErrorMessage report(ParserTags tag) const {
213             return ErrorMessage(tag, stderr, errorFlag);
214         }
215
216         tree_parse_info<iterator_t, factory_t> parseString(const char* s, string
217             iterator_t begin(s, s+strlen(s), filename);
218             iterator_t end;
219             SkipGrammar skip;
220             return ast_parse<factory_t>(begin, end, *this, skip);
221         }
```

```
222
223     private:
224         std::ostream& stderr;
225         bool& errorFlag;
226 };
227
228 #endif
```

Listing A.7: SkipGrammar.h

```
1  /*
2   * SkipGrammar.h
3   *
4   *  Created on: 2/08/2009
5   *      Author: timothyc
6   */
7
8  #ifndef SKIP_GRAMMAR
9  #define SKIP_GRAMMAR
10
11 #include <string>
12 #include <sstream>
13
14 #include "GrammarUtility.h"
15
16 class SkipGrammar : public grammar<SkipGrammar> {
17     public:
18         template <typename ScannerT>
19         class definition {
20             public:
21                 typedef ScannerT scanner_t;
22                 definition(SkipGrammar const& self) {
23                     skip
24                     = +space_p
25                     | comment_p("//")
26                     | comment_p("/*", "*/")
```

```
27                  ;
28
29              }
30
31              rule<scanner_t, parser_tag<skip_tag> > const& start() const {
32                  return skip;
33              }
34
35          private:
36              rule<scanner_t, parser_tag<skip_tag> > skip;
37      };
38
39  };
40
41  #endif
```

Appendix B

Jasmin Syntax

Class definition

.class full/package/name/classname

Where to put the output class; the slash-separated package name corresponds to the directory structure as per the Java-paradigm.

comment

$\backslash s$ *;* ... $\backslash n$ (that is: whitespace semicolon comment newline)

A comment, that will not appear in the bytecode. Note that comments must be preceded by whitespace and extend to the end of line.

directive

.directive

Meta information for the JVM; not part of user code. The available directives are:

- *.catch* add exception handler to a range of code
- *.class* for defining Java classes (described above)
- *.end* end a class or method definition
- *.field* define a class variable
- *.implements* specify an interface that the enclosing class definition implements
- *.interface* for defining Java interfaces (counterpart to *.class*)
- *.limit* assert limits on the enclosing function (for the JVM's benefit)
- *.line* correlate bytecode instructions with source line numbers to help debuggers
- *.method* define a class method
- *.source* the source file used to produce this assembly
- *.super* specify the super class of the enclosing class
- *.throws* declare that the enclosing function may throw exceptions
- *.var* define a local variable's name and scope (to help out debuggers)

function call

invokevirtual package/class/method(arguments)return

Make (invoke) a (virtual/polymorphic) function call.

load field from class

getfield package/class/field type

Push class instance field value onto stack.

load static field

getstatic package/class/field type

Push static class field value onto stack.

instruction

instruction [parameters]

Java mnemonics.

label

name:

A named location; the target of jump and branch statements. May not start with a digit, cannot contain the symbols '=', ':', '.', '"' or '-'. Labels can only be declared inside method definitions, which they are local to.

local variable space

.limit locals number

The maximum number of local variables and function parameters that the current method can hold on the stack. Larger datatypes (long, double) count as two.

operand space

.limit stack number

The maximum number of bytecode operands (instructions) that the current method can hold on the stack at any one time. Instructions are one byte, as are most types. Larger datatypes (long, double) count as two.

store field from class

putfield package/class/field type

Pop value off stack and store in a class instance field.

store static field

putstatic package/class/field type

Pop value off stack and store in a static class field.

source file

.source classname.java

The source file (for debugging support). Use the plain source name without any package/directory prefixes (use String.java, not java/lang/String.java).

statement

directive | *instruction* | *label*

The body of each *Jasmin* function definition is populated with zero or more 'statements' – JVM directives, literal instructions or labelled points in code.

Appendix C

Boost::spirit Syntax

Boost::spirit is a parser library that operates on the idea of 'parsers' – small objects that can recognise simple regular expressions. For example, an *int_p* parser recognises an unbroken sequence of digits, while a *str_p* parser will recognise a specific user-provided string of characters. These simple parsers can then be combined into much larger and more sophisticated parsers via special operators for sequences, non-deterministic iteration (Kleene star) or non-deterministic selection, amongst others. These operators have been chosen such that they resemble those found in extended Backus-Naur form – a formal grammar notation – and are also available for user-overloading.

An older but more complete version of *boost::spirit* was used for this thesis due to issues with availability and documentation. This version is now known as *spirit classic*. A newer version of *spirit* called $V2$ will eventually supercede this implementation. However installation candidates are still uncommon and documentation is too sparse for it to have been worth considering. At the time of writing, the *classic* documentation was sufficient to produce a working parser, though some experimentation was required.

C.1 Parsers

There are several primitive parsers available as part of the *spirit* distribution. The most useful ones are introduced here:

- *space_p*: matches one whitespace character (space, tab, newline, ...) in the input text
- *alpha_p*: matches one alphabetic character in the input text
- *alnum_p*: matches one alphanumeric character in the input text

- *ch_p(c)*: matches c in the input text
- *str_p(userString)*: matches *userString* in the input text
- *int_p*: matches an integer in the input text
- *eps_p*: always matches, but doesn't match against anything. This parser is most useful for triggering semantic actions for error reporting.
- *nothing_p*: never matches (forces this branch to fail and the parser to backtrack)
- *comment_p(marker)*: matches input text between *marker* and the end of the current line
- *comment_p(start, end)*: matches input text between *start* and *end* (possibly across multiple lines)
- *list_p(a, b)*: matches a list of a parsers interleaved by b parsers, for example *ababa* but not *abb*. This is a convenience parser that can be handwritten as: $a >> *(b >> a)$

Further discussion of these parsers is left to the *spirit* documentation, which is reasonably complete on this topic.

C.2 Operators

Boost::spirit provides operators for combining parsers. Many of these operators resemble regular expression operators:

- ! for zero or one
- $*$ for zero or more
- $+$ for one or more

Of more interest are the $>>$ and $|$ operators. The $>>$ operator is used for matching a sequence of parsers. For example, $a >> b$ matches some input text if a matches the first section and b matches the rest. Meanwhile the 'or' operator $|$ allows *spirit* to try one parser, and then try an alternative if that fails. For example:

$$ch\_p('a') \mid 'b' \mid 'c'$$

which will match either the character 'a', 'b' or 'c' depending on the input text.

Finally, attention is directed to the 'ch_p' prefix; *boost::spirit* overloads most parsers so that they will accept other parsers. However this requires at least one *spirit* parser[1]. Ch_p is one such parser – which overloads the | operator – and is capable of accepting literal characters as if they were also ch_p parsers. This notation is sugar[2] for the more explicit:

$$ch\_p('a') \mid ch\_p('b') \mid ch\_p('c')$$

C.3 Grammars

Parsers are useful for building up larger parsers which can be assigned to 'rules'. Rules can in turn talk about each other. For example, a simplified subset of the *Jolog* grammar appears as:

$$\begin{aligned} identifier &= (alpha\_p \mid '\_') >> *(alnum\_p \mid '\_'); \\ typeName &= identifier; \\ variable\_definition &= typeName >> identifier >> !('=' >> expression); \end{aligned}$$

However too many rules forces the *spirit*-based components of a program to take a *long* time to compile. The solution is to wrap related groups of rules into 'grammars'. Grammars can then be used within rules as before, such as the use of *expression* in the above example.

Grammars are created in dedicated classes; the boilerplate for an example grammar may appear as:

Listing C.1: an example *boost::spirit* grammar

```
1 class ExampleGrammar : public grammar<ExampleGrammar> {
2     public:
3         template <typename ScannerT>
4         class definition {
5             public:
6                 definition(ExampleGrammar const& self) {
7                     //definition of rules
8                     exampleRule = str_p("example") >> otherGrammar;
```

[1]This is due to 'limitations' in *C++* overloading: you cannot overload *operator*|(*char*, *char*) but you can overload *operator*|(*myParser*, *char*).

[2]Indeed those who have an aversion to excessive operator overloading may view this as a little too sweet for their tastes.

```
9
10              }
11
12              //what rule does this grammar correspond to
13              rule<scanner_t, parser_tag<example_tag> > const& start() const {
14                  return exampleRule;
15              }
16
17          private:
18              //we need to store other grammars if we use them
19              MyOtherGrammar otherGrammar;
20
21              //type definitions for each rule
22              rule<Scanner_t, parser_tag<example_tag> > exampleRule;
23      };
24  };
```

Note in particular that our class (*ExampleGrammar*) is simply a shell for a required inner class called *definition* (whose constructor expects a reference to our outer class). Note also that our class extends the *spirit* class *grammar* which is also parameterised in terms of our own shell class. The implications of this bizarre inheritance hierarchy are unimportant for our purposes – they are simply an artifact of the metatemplate implementation. However the implications of debugging simple errors in this setup manifest themselves in hundreds of lines of error output[3].

C.4 Skip Grammar

These simple rules are fine for examples, but a *Java*-like grammar is a large, complicated beast and error checking obfuscates this further. The task of adding whitespace independence to the grammar appears daunting in this light. *Boost::spirit*, however, provides a convenient solution to this problem: rather than adding a rule such as $*space\_p$ between *every* other rule, we can instead provide a 'skip' grammar to the parsing function.

We have already seen a simple skip grammar: $*space\_p$. A more complicated *C*-like parser can

[3]This is no exaggeration. Accidental omission of a semicolon or a function parameter can literally produce an error message that spans several screens.

be constructed by adding support for comments with the *comment_p* parser:

$$*(space\_p \mid comment\_p(\text{“//”}) \mid comment\_p(\text{“/*”},\ \text{“*/”}))$$

The above rule can be provided directly to the parsing function:

$$ast\_parse(begin,\ end,\ main\_grammar,\ *(space\_p \mid comment\_p(\text{“//”}) \mid comment\_p(\text{“/*”},\ \text{“*/”})))$$

Alternatively, we can encapsulate the above rule inside a proper *spirit* grammar. The grammar itself can then be provided to the parse function in place of the above rule:

$$ast\_parse(begin,\ end,\ main\_grammar,\ skip\_grammar)$$

C.5 Directives

After going to the effort of essentially turning whitespace off, we may find ourselves in a situation where whitespace is particularly important – inside a string literal or between digits in a number for example. In these cases we want to explicitly indicate that whitespace cannot occur (or occur in quantities that we accept by explicitly adding a *space_p* parser). We can achieve this by wrapping our subrule within a *lexeme_d* directive, which disables the skip grammar within its bounds. A correct implementation of the above ‘identifier’ rule may then appear as:

$$identifier = lexeme\_d[\ (alpha\_p \mid '\_')\ >>\ *(alnum\_p \mid '\_')\ ];$$

Two other important directives are used for structuring abstract syntax trees: the *root_node_d* directive indicates that its contents will become the root of the current subtree, *token_node_d* ensures that its contents become a single child of the current subtree (or the root if no competitors exist!) and *no_node_d* instructs *spirit* to match its parser against the input text, but to discard the match rather than add it to the AST. The *root_node_d* and *no_node_d* have obvious uses in structuring the tree to a more useful form for later use. The *token_node_d* is a little more subtle: it is most useful when used to wrap a *lexeme_d* directive since it will join the individual characters that *lexeme_d* returns into a proper string with specific whitespace properties. This means that an AST node for a string literal can be a single node, rather than a parent node with one child for each character.

References

[1] H. J. Levesque, R. Reiter, Y. Lesprance, F. Lin, and R. B. Scherl, "GOLOG: A logic programming language for dynamic domains," *Journal of Logic Programming*, vol. 31, 1997.

[2] G. D. Giacomo, Y. Lesprance, and H. J. Levesque, "Congolog, a concurrent programming language based on the situation calculus," 2000, available at http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=3772EFE9F5B77F0D5156B3542FE3AB3D?doi=10.1.1.34.9024&rep=rep1&type=pdf [last accessed 25 Sep].

[3] J. Funge, "CML documentation," 1998,
available (via Wayback Machine) at http://web.archive.org/web/20040202190634/www.dgp.toronto.edu/~funge/cml.html [last accessed 21 May].

[4] J. Funge, X. Tu, and D. Terzopoulos, "Cognitive modeling: knowledge, reasoning and planning for intelligent characters," in *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 29–38.

[5] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, vol. 2, no. 3-4, pp. 189–208, 1971. [Online]. Available: http://dx.doi.org/10.1016/0004-3702(71)90010-5

[6] D. V. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "PDDL – the planning domain definition language," Yale Center for Computational Vision and Control, Tech. Rep. 003, 1998,
available at ftp://ftp.cs.yale.edu/pub/mcdermott/software/pddl.tar.gz [last accessed 23 May].

[7] P. Eyerich, B. Nebel, G. Lakemeyer, and J. Claßen, "GOLOG and PDDL: what is the relative expressiveness?" in *PCAR '06: Proceedings of the 2006 international symposium on Practical cognitive agents and robots*. New York, NY, USA: ACM, 2006, pp. 93–104.

[8] M. Thielscher, S. Schiffel, C. Drescher, M. Fichtner, and Y. Martin, "FLUX," 2006, available at http://www.fluxagent.org/home.htm [last accessed 23 May].

[9] N. McCain, "Causal calculator," 1997, available at http://www.cs.utexas.edu/~tag/cc/ [last accessed 23 May].

[10] R. J. Firby, "Adaptive execution in complex dynamic worlds," Ph.D. dissertation, Yale, 1989, available (via Google cache) at http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=D4612BFFD5A2576672F6257B0DD4DC4F?doi=10.1.1.17.9&rep=rep1&type=pdf [last accessed 23 May].

[11] N. J. Nilsson, "Teleo-reactive programs for agent control," *CoRR*, vol. cs.AI/9401101, 1994.

[12] D. Weerasooriya, A. S. Rao, and K. Ramamohanarao, "Design of a concurrent agent-oriented language," 1994, available at http://www2.umassd.edu/SWagents/agentdocs/AAII/technote52.pdf [last accessed 24 May].

[13] A. S. Rao, "AgentSpeak(L): BDI agents speak out in a logical computable language," in *Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, R. van Hoe, Ed., Eindhoven, The Netherlands, 1996. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.1476

[14] J. F. Hübner and R. H. Bordini, "Jason: a java-based interpreter for an extended version of AgentSpeak," 2009, available at http://jason.sourceforge.net/JasonWebSite/Jason%20Home.php [last accessed 24 May].

[15] M. P. Georgeff and A. L. Lansky, "Reactive reasoning and planning," in *AAAI-87 Proceedings*. American Association of Artificial Intelligence, 1987, pp. 677–682, available at http://www.aaai.org/Papers/AAAI/1987/AAAI87-121.pdf [last accessed 24 May].

[16] AOS Group, "Jack: an agent infrastructure for providing the decision-making capability required for autonomous systems," 2009, available at http://www.aosgrp.com/downloads/JACK_WhitePaper_UKAUS.pdf [last accessed 24 May].

[17] A. J. Champandard, "AiGameDev.com," 2008,
available at http://aigamedev.com [last accessed 26 April].

[18] S. Jacobs, "Applying Readylog to Agent Programming in Interactive Computer Games," 2005,
honours thesis, RWTH Aachen University, Germany.
available at http://robocup.rwth-aachen.de/readybot [last accessed 21 May].

[19] FIPA, "Foundation for Intelligent Physical Agents," 2002-2009,
available at http://www.fipa.org/ [last accessed 21 May].

[20] D. R. Jonathan Meyer and I. Kharon, "Jasmin," 2005,
available at http://jasmin.sourceforge.net/ [last accessed 27 Sep].