An Adaptive Behavior Language: Adaptive Programming for Advanced Agent Modeling
CS 6390 Final Project, Fall 2007

Christopher Simpkins
with help from Sooraj Bhat

May 1, 2008
Abstract

Current programming languages and software engineering paradigms are proving insufficient for building intelligent multi-agent systems—such as interactive games and narratives—where developers are called upon to solve increasingly complex and ill-specified problems. A promising solution is to build adaptive systems; that is, to develop software written specifically to adapt to its environment by changing its behavior in response to what it learns about the world. Here we describe a new programming language, An Adaptive Behavior Language (A²BL), that implements adaptive programming primitives to support partial programming. Partial programming enables programmers to more easily encode software agents that are difficult or impossible to write in existing languages that do not offer language-level support for adaptivity. We motivate the use of partial programming with an example agent coded in a cutting-edge, but non-adaptive agent programming language (ABL), and show how A²BL can encode the same agent much more naturally.
# Contents

I Overview ........................................ 3

1 Purpose ........................................ 4

2 Intended Users .................................. 4

3 Design Goals .................................... 5

4 Related Work .................................... 5

4.1 The Path to Adaptive Programming: Integrating Machine Learning into a Programming Language .................................. 6

4.2 The Path to Adaptive Software Engineering: Practical Languages for Large Agent-Based Applications .......................... 6

5 Meeting A$^2$BL’s Design Goals: A$^2$BL as a Model of Adaptive Programming ........................................ 7

5.1 Functions and Classes versus Agents and Modules ........................................ 7

5.2 Input/Output versus Perception/Action ........................................ 7

5.3 Logic-based versus Probability-based ........................................ 7

5.4 Goal-based versus Utility-based ........................................ 8

5.5 Sequential, single- versus Parallel, multi- ........................................ 8

5.6 Hand-programmed versus Trained (Learning) ........................................ 8

5.7 Fidelity to designer versus Perform well in environment ........................................ 8

5.8 Pass test suite versus Scientific method ........................................ 9
II Users Manual

7 A Behavior Language (ABL) ................................................. 12
8 Hello World! A Simple ABL Agent .................................... 13
  8.1 Compiling and Running an ABL Agent ........................... 13
  8.2 A More Detailed Example: The Predator-Food World ....... 15
  8.3 A Predator-Food Agent in ABL ................................. 15
9 An Adaptive Behavior Language (A²BL) .............................. 18
  9.1 The Predator-Food Agent In A²BL .............................. 18

III Reference Manual

10 Background ........................................................................ 22
  10.1 AI Planning ................................................................. 22
  10.2 Machine Learning ........................................................ 22
  10.3 Reinforcement Learning ............................................... 23
  10.4 Modular Reinforcement Learning ................................. 23
11 Syntax and Semantics of A²BL ........................................... 24
12 Error Handling ................................................................ 26
13 A BNF Grammar for A²BL ............................................... 26
Part I

Overview
0.1 Purpose

The purpose of A²BL, is to enable the writing of adaptive software agents. By adaptive software we refer to the notion of adaptive used in the machine learning community: software that learns to adapt to its environment during run-time, not software that is written to be easily changed by modifying the source code and recompiling. In particular, we use Peter Norvig’s definition of adaptive software:

> Adaptive software uses available information about changes in its environment to improve its behavior [Norvig and Cohn(1998)].

0.2 Intended Users

Because “behavior” is a part of the ABL acronym, one might believe that ABL is designed for experts in human behavior, such as psychologists or sociologists. While ABL can support the needs of such designers, ABL is a complex language that exposes many technical details to agent authors, making it suitable mainly for programming experts. So far, mainly senior undergraduate and graduate students in computer science have been productive with ABL. Given that we envision A²BL as a tool for non-programming experts, and A²BL is based on ABL, we must consider several important questions:

- What kinds of abstractions and language features are required by behavior experts such as psychologists to effectively encode their domain knowledge in A²BL?

- Can such non-programmer-oriented language features subsume the advanced features that lead to ABL’s complexity without losing the power they bring to ABL?

- Noting Alan Perlis’s epigram—“a programming language is low level when its programs require attention to the irrelevant”—what is irrelevant when modeling intelligent agents?

- Is it desirable to have both programmer-oriented, and domain expert-oriented language features in A²BL so that an agent author can choose to “get down and dirty” sometimes and maintain a higher level of abstraction at other times?

- Is it realistic to expect psychologists or sociologists to adopt a form of computer programming as a basic part of their methodological tool kit? How should we go about making that happen?

While the questions above represent significant issues to be tackled during the long development of A²BL, here we focus on the addition of adaptivity to the language, which we believe will go a long way towards making A²BL usable for non-programming experts.
0.3 Design Goals

The overarching design goal of A²BL is to enable our intended users—non-programming experts—to write significant adaptive software agents. To this end, we adopt Peter Norvig’s model of adaptive programming as our design goals for A²BL. Norvig identifies several requirements of adaptive software—adaptive programming concerns, agent-oriented concerns, and software engineering concerns—and five key technologies—dynamic programming languages, agent technology, decision theory, reinforcement learning, and probabilistic networks—needed to realize adaptive software. These requirements and technologies are embodied in his model of adaptive programming given in Table 1.

<table>
<thead>
<tr>
<th>Traditional Programming</th>
<th>Adaptive Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function/Class</td>
<td>Agent/Module</td>
</tr>
<tr>
<td>Input/Output</td>
<td>Perception/Action</td>
</tr>
<tr>
<td>Logic-based</td>
<td>Probability-based</td>
</tr>
<tr>
<td>Goal-based</td>
<td>Utility-based</td>
</tr>
<tr>
<td>Sequential, single</td>
<td>Parallel, multi-</td>
</tr>
<tr>
<td>Hand-programmed</td>
<td>Trained (Learning)</td>
</tr>
<tr>
<td>Fidelity to designer</td>
<td>Perform well in environment</td>
</tr>
<tr>
<td>Pass test suite</td>
<td>Scientific method</td>
</tr>
</tbody>
</table>

Table 1: Peter Norvig’s model of adaptive programming [Norvig(1998)].

A²BL integrates two of Norvig’s key technologies: agent technology and reinforcement learning. We will explain how A²BL implements Norvig’s adaptive programming model and argue that A²BL satisfies many of Norvig’s requirements, with the rest slated for future development. Before we proceed, we discuss related work in integrating reinforcement learning into programming languages and expand on Norvig’s view of the role of machine learning in general, and reinforcement learning (RL) in particular in the realization of adaptive programming, and

0.4 Related Work

There is already a body of work in integrating reinforcement learning into programming languages, mostly from Stuart Russell and his group at UC Berkeley [Andre and Russell(2001), Andre and Russell(2002)]. Their work is based on hierarchical reinforcement learning [Parr and Russell(1998), Dietterich(1998)], which enables the use of prior knowledge by constraining the learning process with hierarchies of partially specified machines. This formulation of reinforcement learning allows a programmer to specify parts of an agent’s behavior that are known and understood already while allowing the learning system to learn the remaining parts in a way that is consistent with what the programmer specified explicitly.
The notion of programmable hierarchical abstract machines (PHAM) [Andre and Russell(2001)] was integrated into a programming language in the form of a set of Lisp macros (ALisp) [Andre and Russell(2002)]. Andre and Russell provided provably convergent learning algorithms for partially specified learning problems and demonstrated the expressiveness of their languages, paving the way for the development of RL-based adaptive programming. Our work builds on theirs but with a focus on practical applications.

**0.4.1 The Path to Adaptive Programming: Integrating Machine Learning into a Programming Language**

One of the promises of machine learning is that it allows designers to specify problems in broad strokes while allowing a machine to do further parameter fine-tuning. Typically, one thinks of building a system or agent for some specific task and then providing it some kind of feedback, allowing it to learn. In this case, the agent is the point of the exercise. A²BL embeds this notion within a programming language itself by extending it with adaptive behaviors. The power of such a merger of machine learning and a programming language is that it allows for what has become known as *partial programming*; that is, it allows a designer to specify what he knows how to express exactly and leave the system to learn how to do the rest.

**0.4.2 The Path to Adaptive Software Engineering: Practical Languages for Large Agent-Based Applications**

We have chosen another language, ABL (which we shall describe in some detail later), as the starting point for our adaptive programming language because ABL is designed for developing intelligent autonomous agents for significant end-user applications, namely games and interactive narratives. A²BL serves two purposes. First, with a modular implementation of adaptive behaviors that enables the swapping of RL algorithms, A²BL provides a platform for RL research. Second, A²BL is the first step towards a language that supports the needs of game designers and social science modelers writing practical, large scale agent systems. It is the second purpose, the practical purpose, that distinguishes our work from previous work in RL-based adaptive programming.
0.5 Meeting A²BL’s Design Goals: A²BL as a Model of Adaptive Programming

In the introduction, we listed the elements of Peter Norvig’s model of adaptive programming [Norvig(1998)], which represent the design goals of A²BL. Here we discuss A²BL’s implementation of this model.

0.5.1 Functions and Classes versus Agents and Modules

A²BL inherits the agent-orientation of ABL. The fundamental units of abstraction are agents and behaviors, where an agent is essentially a collection of behaviors. One could think of agents as analogous to classes/objects and behaviors as analogous to functions, but the analogy quickly breaks down. First, agents can’t be composed of other agents the way objects can be composed of other objects. Second, functions are called directly in a procedural fashion; behaviors are specified declaratively and selected for execution by ABL’s runtime planning system only if and when those behaviors are needed to pursue some goal. ABL’s declarative reactive planning paradigm, and A²BL’s adaptive model provide much better support for a style of programming that separates the what of agent behavior from the how.

0.5.2 Input/Output versus Perception/Action

In traditional programming, even to a large extent in event-driven object-oriented programming, programs are written and reasoned about in terms of input/output behavior. A function is given some input and produces some output. A class is given responsibility for some part of the application’s data, responds to particular messages, and provides particular responses. In agent-oriented programming, on the other hand, the agent programmer thinks in terms of what an agent can perceive in the world, and what actions the agent can execute to modify the state of the world. In ABL and A²BL, perception is modeled by WMEs that represent the agent’s awareness of the world in which it is situated. Actions are procedural calls within behaviors that effect changes in whatever world the agent is operating in. The WME (perceptions) and actions constitute an API between agents and worlds, effectively decoupling agents and worlds.

0.5.3 Logic-based versus Probability-based

In traditional programming, selection logic (boolean tests and if/then constructs) is an important part of any non-trivial program. To a large extent, this is true even in ABL, where
behaviors are selected based on logical preconditions. By integrating RL, A²BL incorporates probabilistic reasoning into the core of the language: RL algorithms build probabilistic models of the world and of agent optimal behavior in that world. In this way, A²BL provides explicit support for probabilistic reasoning without the programmer having to think explicitly about stochasticity.

0.5.4 Goal-based versus Utility-based

Goal attainment is a fundamental metaphor in ABL, and in agent programming in general. In A²BL, goal attainment is represented explicitly in terms of rewards, or utilities. Every state in the world has an associated utility (often implicitly zero), and A²BL’s adaptive features seek to maximize the agent’s utility automatically.

0.5.5 Sequential, single- versus Parallel, multi-

A²BL inherits ABL’s parallelism and extends it to support concurrent modular reinforcement learning.

0.5.6 Hand-programmed versus Trained (Learning)

With A²BL’s support for partial programming, the programmer can ignore low-level behavior that is either too poorly specified or too dynamic to encode explicitly and leave A²BL to learn the details.

0.5.7 Fidelity to designer versus Perform well in environment

In traditional software engineering, a program is good if it conforms to its specification. In adaptive partial programming, a program is good if it performs well in whatever environment it finds itself in. With A²BL’s explicit support for reward and state specification, and its automatic learning of policies, A²BL agents are written to perform well in their environments even when design specifications are vague.
0.5.8 Pass test suite versus Scientific method

Closely related to the previous point, test suites are written to test a program’s conformance to design specifications; however, a certain amount of experimentation is often necessary to determine just what exactly is the right thing to do in given situations. Yet there is always some imperative to act given whatever information you have at the moment. As a technical matter, reinforcement learning makes explicit this tradeoff between the exploration of environments and the exploitation of already gained knowledge. A²BL inherits this principled approach to the exploration/exploitation tradeoff by using RL to implement adaptivity.

0.6 Research Issues and Future Directions

Currently, we have implemented an ANTLR-based parser for A²BL, and we have tested several reinforcement learning algorithms for use in A²BL agents. The major remaining task in implementing A²BL, and by far the major portion of the work, is to integrate the reinforcement learning algorithms with the code generation phase of the A²BL compiler.

Many challenging and important issues need to be addressed to fully realize our vision for A²BL. These issues range from foundational RL theory to pragmatic software engineering considerations. We discuss some of the major issues below.

0.6.1 Adaptive Software Engineering

At the end of the day, an agent is a kind of computer program running in a run-time environment. Whatever language features A²BL supports, computer programs (or computational agent models) will need to be written and debugged. Given the complexity of individual agents and our desire to support real world-scale multi-agent system modeling, the task of writing A²BL agents and multi-agent systems is likely to be a significant effort, akin to that of a large software engineering project. We will therefore need to address many of the same issues as traditional software engineering:

- Are there effective visual metaphors for agent behavior that would enable the effective use of a visual programming environment for A²BL?
- What does it mean to “debug” an intelligent agent or multi-agent system?
- Can some of the mechanisms for structuring large software systems, such as objects and modules, be transferred effectively to an agent-authoring domain? What new kinds of structuring mechanisms need to be invented?
• Can the A²BL language, compiler, and run-time environment be designed in such a way that the agent author need not be concerned with efficiency or optimization? If not, are we resigned to requiring expert programmers to author intelligent agents?

0.6.2 OOP in A²BL

ABL does not currently support inheritance. It seems natural to model agents with an inheritance hierarchy similar to OO modeling in modern software engineering; however, supporting inheritance in agents may not be as simple as borrowing the body of existing theory from OOP. Agents are more than objects, and their behavior is stochastic. What would it mean for an agent to be a subtype of another agent? Would we call this an “is-a” relationship? Would we ascribe all of the semantics that OOP ascribes to “is-a” relationships? In particular, how do we model preconditions and postconditions in a stochastic agent? Because type inheritance, or some related form of reuse, seems useful for supporting large-scale, real-world agent programming, it is worthwhile to develop the theory necessary to implement an inheritance mechanism that (1) supports the design of large systems of agents and (2) supports reuse mechanisms for A²BL.
Part II

Users Manual
A2BL is essentially ABL with five additional constructs for adaptivity. Thus, we begin with a discussion of ABL.

0.7 A Behavior Language (ABL)

ABL (A Behavior Language) represents the cutting edge of implemented agent modeling languages [Mateas and Stern(2004)]. ABL is a reactive planning language with Java-like syntax based on the Oz Project believable agent language Hap [Loyall and Bates(1991)]. It has been used to build actual live interactive games and dramas, such as Facade [Mateas and Stern(2003)]. In Facade, developed by Andrew Stern and Michael Mateas, the player is asked to deal with a relationship between an arguing couple. It is a single act drama where the player must negotiate her way through a minefield of personal interactions with two characters who just so happen to be celebrating their ten-year marriage.

An ABL agent consists of a library of sequential and parallel behaviors with reactive annotations. Each behavior consists of a set of steps to be executed either sequentially or in parallel. There are four basic step types: acts, subgoals, mental acts and waits. Act steps perform an action in the world; subgoal steps establish goals that must be accomplished in order to accomplish the enclosing behavior; mental acts perform bits of pure computation, such as mathematical computations or modifications to working memory; and wait steps can be combined with continually-monitored tests to produce behaviors that wait for a specific condition to be true before continuing or completing.

The agent dynamically selects behaviors to accomplish specific goals and attempts to instantiate alternate behaviors to accomplish a subgoal whenever a behavior fails. The current execution state of the agent is captured by the active behavior tree (ABT) and working memory. Working memory contains any information the agent needs to monitor, organized as a collection of working memory elements (WMEs). There are several one-shot and continually-monitored tests available for annotating a behavior specification. For instance, preconditions can be written to define states of the world in which a behavior is applicable. These tests use pattern matching semantics over working memory familiar from production rule languages; we will refer to them as WME tests.

To motivate the discussion of A2BL we will discuss the development of agents in ABL and point out the issues with writing agents in ABL, and show how A2BL addresses these issues. In the User’s Manual, we will then implement the same agent using A2BL to show the benefits to the programmer of integrating true adaptivity into the programming language itself.
0.8 Hello World! A Simple ABL Agent

This tutorial will show you the basic structure of an ABL source code file and how to compile and run ABL source code. The goal is to ensure that you have ABL installed correctly. We assume some familiarity with Java. ABL is very Java-like, and we will point out where the two are similar and where they differ. You will need Java 1.5 to compile and run ABL code, but currently only Java 1.4 syntax is allowed inside mental acts.

The following is the complete code for a simple ABL agent, contained in a file named HelloWorld.abl:

```java
package test;

behaving_entity HelloWorld {
    initial_tree {
        mental_act {
            System.out.println("Hello world!");
        }
    }
}
```

An ABL source code file must be named AgentName.abl, where AgentName is the name used in the behaving_entity declaration. An agent must have a package declaration; this is a limitation of the current version of ABL.

Though the analogy is false, you can think of ABL behaving entities as Java classes and ABL behaviors as Java methods. Likewise, the initial_tree can be thought of as the agent’s main() method. The initial tree controls the main flow of behaviors in an ABL agent, and every ABL agent has one. The initial tree is just a behavior, and like any behavior, it contains a set of steps. In this case, it has a single step, which is a mental act that prints a message. Normally, agents have sensors and actions. In this case, we have an agent who has neither because we did not register any. Thus, we are left with an agent that can only perform mental acts; i.e. it can "think" but can’t "see" or "do".

0.8.1 Compiling and Running an ABL Agent

Assume HelloWorld.abl lives in a directory named test and that we are working from the directory containing test. The following steps compile and run this agent.

1. Run the ABL compiler on AgentName.abl

You can invoke the ABL compiler as follows:
where ABLPATH is an environment variable that contains items needed to run the ABL compiler or items that the ABL compiler will need to resolve certain references in your ABL code. For convenience, we have included a helper script which enables you to type the following instead:

```
abl test/HelloWorld.abl
```

This works provided that (1) abl.jar is on the ABLPATH; and (2) the abl script is on the path.

2. Compile the resultant Java files.

Running the ABL compiler will generate a dozen or so Java files in the same package. Compile them:

```
javac -cp $ABLPATH:. test/HelloWorld*.java
```

Typically, you will have other code that your agent refers to, so you will need to include them on your classpath as needed.

3. Invoke AgentName.startBehaving().

An agent AgentName.abl will generate a file AgentName.java, which will have a method startBehaving(). Call this method to get your agent to start executing its flow of behavior described in initial_tree. This method must be invoked in the same thread in which the agent is constructed/new’ed. A safe invocation pattern is thus:

```
new HelloWorld().startBehaving();
```

For convenience, AgentName.java contains a simple main() that does exactly this. So you could invoke the HelloWorld agent as follows:

```
java -cp $ABLPATH:. test.HelloWorld
```

Again, we have included a helper script, abl, that does exactly this and saves you some typing:

```
abl test.HelloWorld
```

This simple main() probably won’t be suitable for more complex agents, as you will typically need to do some initialization of the sensorimotor system first. This depends on your sensorimotor system though; it is possible to do all initialization in static initializers, and this may be appropriate for sensorimotor systems of medium to low complexity.
Note that while the sensory-motor architecture (SMA) of ABL is important for writing significant agents, explaining it in sufficient detail would require much more space and would distract from the core issue: adaptivity. Thus, we direct interested readers to [Mateas and Stern(2004)] and proceed with an example that illustrates the issues of adaptivity without getting bogged down with SMA issues.

0.8.2 A More Detailed Example: The Predator–Food World

We will analyze two different implementations of an agent for the Predator–Food world. The Predator–Food world is a grid where there are two main activities: avoiding the predator and finding food. At every time step, the agent must pick a direction to move. Food appears randomly at fixed locations, and there is a predator in the environment who moves towards the agent once every other time step.

0.8.3 A Predator–Food Agent in ABL

Figure 1 presents an ABL agent for the Predator–Food world.

The following code snippet defines a an agent and its principle behavior, LiveLongProsper.

```
behaving entity FurryCreature {
    parallel behavior LiveLongProsper() {
        subgoal FindFood();
        subgoal AvoidPredator();
    }
    // ...

    LiveLongProsper is defined as a parallel behavior to reflect the fact that both of its subgoals must be pursued in parallel in order for the enclosing behavior to succeed. In the following code snippet, the FindFood subgoal is defined as a sequential behavior.

    sequential behavior FindFood() {
        with (ignore_failure) subgoal MoveNorthForFood();
        with (ignore_failure) subgoal MoveSouthForFood();
        with (ignore_failure) subgoal MoveEastForFood();
        with (ignore_failure) subgoal MoveWestForFood();
    }

    Each of the subgoals—MoveNorthForFood, MoveSouthForFood, MoveEastForFood, and MoveWestForFood—must be performed in a particular sequence if the agent is to succeed in finding food. Note that, because some subgoals will not be selected for execution in any given time step, the subgoals must be annotated with ignore_failure to prevent the enclosing behavior from failing. The agent will only move in one direction in each time step, so three of the subgoals will fail because their preconditions will not be satisfied. Following the reactive planning paradigm of ABL, MoveNorthForFood is defined as follows:
behaving_entity FurryCreature
{
    parallel behavior LiveLongProsper()
    {
        subgoal FindFood();
        subgoal AvoidPredator();
    }

    // subgoal 1
    sequential behavior FindFood()
    {
        with (ignore_failure) subgoal MoveNorthForFood();
        with (ignore_failure) subgoal MoveSouthForFood();
        with (ignore_failure) subgoal MoveEastForFood();
        with (ignore_failure) subgoal MoveWestForFood();
    }

    // subgoal 2
    sequential behavior AvoidPredator()
    {
        with (ignore_failure) subgoal MoveNorthAwayFromPredator();
        with (ignore_failure) subgoal MoveSouthAwayFromPredator();
        with (ignore_failure) subgoal MoveEastAwayFromPredator();
        with (ignore_failure) subgoal MoveWestAwayFromPredator();
    }

    sequential behavior MoveNorthForFood()
    {
        precondition {
            (FoodWME x::foodX y::foodY)
            (SelfWME x::myX y::myY)
            ((foodY - myY) > 0) // The food is north of me
        }

        // Code for moving agent to the north elided
    }

    // ...

    sequential behavior MoveNorthAwayFromPredator()
    {
        precondition {
            (PredatorWME x::predX y::predY)
            (SelfWME x::myX y::myY)
            (moveNorthIsFarther(myX,myY,predX,predY))
        }

        // Code for moving agent to the north elided
    }
}

Figure 1: An ABL agent for the Predator-Food world.

The precondition block defined at the beginning of the behavior defines the circumstances under which ABL’s run-time planning system may select this behavior for execution. It basically says that if the food is north of the agent’s position, the agent should move north. This set of preconditions defines the desirability of moving north in search of food, but ignores the predator. The behavior of moving north away from the predator is defined as
follows:

```java
sequential behavior MoveNorthAwayFromPredator() {
    precondition {
        (PredatorWME x::predX y::predY)
        (SelfWME x::myX y::myY)
        (moveNorthIsFarther(myX, myY, predX, predY))
    }

    // Code for moving agent to the north elided
}
```

As in the `MoveNorthForFood` behavior, the conditions under which `MoveNorthAwayFromPredator` may be selected for execution are defined in a `preconditions` block. Note that we have factored the code for computing whether the precondition has been met into a utility function, `moveNorthIsFarther`. Similar subgoal behavior would be defined for each direction of movement, and for each reason for such movement. The full code (with unimportant details elided) is given in Figure 1.

While ABL’s reactive-planning paradigm and declarative system make it possible to define complex autonomous agents, there are several problems. First, each subgoal behavior assumes that the positions of both the food and the predator are known. Second, if there is a conflict between subgoals, the programmer must write code to resolve this conflict. For example, what should the agent do if the FindFood subgoal wants to move north to get to the food, but the AvoidPredator subgoal wants to move south to get away from the predator?

The biggest problem with this ABL agent is that low-level agent actions (movement) and the reasons for selecting those actions are coupled. Because of this coupling, movement behaviors must be duplicated for each possible reason the movement might be executed. Thus, moving north for food and moving north to avoid the predator must be represented separately and the preconditions for each carefully specified. While the movement action itself could be factored into a separate function called by each behavior, there is still a considerable cognitive burden on the programmer who must consider each combination of agent action and reason for action. Note that any programming language that does not provide a means for separating the concerns of what must be done and how it is to be accomplished will impose a similar cognitive burden on agent programmers.

Another problem with the ABL version of the Predator-Food agent is that the programmer must fully specify the agent’s behavior. If there is a part of the agent’s behavior that the programmer does not know, he must implement his best guess. This becomes difficult in the typically ill-specified and dynamic environments where we would want to deploy intelligent agents, such as massively multi-player games.

As we will see in the next sections, integrating adaptivity into the programming language not only reduces the amount of code required to implement an agent, but more importantly allows the programmer to think about what the agent’s goals are, leaving the agent to figure out how to achieve them. This separation of concerns is enabled by partial programming, in which the programmer need only specify what he knows, leaving the run-time system to
0.9 An Adaptive Behavior Language ($A^2$BL)

Our solution to the problems described in the previous section is to provide built-in language support for adaptivity. In $A^2$BL, adaptivity is achieved by integrating reinforcement learning directly into the language.

One could think of $A^2$BL as ABL++. At the very least, $A^2$BL is ABL + Modular Reinforcement Learning (MRL – see Reference Manual). In order to support MRL, $A^2$BL extends ABL with several key concepts.

0.9.1 The Predator–Food Agent In $A^2$BL

In section 8.3 we showed a Predator–Food agent coded in ABL. The ABL code for this agent had to deal with many low-level issues of action selection, essentially hand-coding a policy. In this section we show that, with adaptivity built into the language, it is possible for the programmer to think at a much higher level, reducing the cognitive burden significantly.

Using the state, reward, and action model of reinforcement learning, the programmer can simply say “these are the agent’s goals (in terms of rewards), and these are the actions available to achieve these goals.” The reinforcement learning system learns the states under which given actions should be selected.

The full code (minus irrelevant details of movement implementation) is given in Figure 2. The first difference between the ABL agent and the $A^2$BL agent is that the principal enclosing behavior, LiveLongProsper is defined as an adaptive collection behavior. This tells the $A^2$BL run-time system to treat the enclosed adaptive behaviors as sub-agents in the MRL framework. Each subgoal behavior then defines a set of relevant actions, and the action set of the agent as a whole is the union of all subgoal action sets. Note that each subgoal contains exactly the same actions. There is no need to define different subgoals and the conditions under which they are selected – the learning algorithms built into $A^2$BL automatically handle these tasks.

The goal of a behavior is defined by a reward construct. In the FindFood behavior, the following code defines the goal:

```plaintext
reward {
  100 if { (FoodWME) }
}
```

The code above says that, if the agent finds the food, it gets a large positive reward. Recall
behaving_entity FurryCreature
{
    adaptive collection behavior LiveLongProsper()
    {
        subgoal FindFood();
        subgoal AvoidPredator();
    }

    // subgoal 1
    adaptive sequential behavior FindFood()
    {
        reward
        {
            100 if { (FoodWME) }
        }

        state
        {
            (FoodWME x::foodX y::foodY)
            (SelfWME x::myX y::myY)
            return (myX,myY,foodX,foodY);
        }

        subgoal MoveNorth();
        subgoal MoveSouth();
        subgoal MoveEast();
        subgoal MoveWest();
    }

    // subgoal 2
    adaptive sequential behavior AvoidPredator()
    {
        reward
        {
            -10 if { (PredatorWME) }
        }

        state
        {
            (PredatorWME x::predX y::predY)
            (SelfWME x::myX y::myY)
            return (myX,myY,predX,predY);
        }

        subgoal MoveNorth();
        subgoal MoveSouth();
        subgoal MoveEast();
        subgoal MoveWest();
    }

    // ...
}

Figure 2: An A^2BL agent for the Predator-Food world.

that WMEs are the mechanism by which an agent senses the world in ABL (and in A^2BL). Now consider the state definition for FindFood:

state
{
    (FoodWME x::foodX y::foodY)
    (SelfWME x::myX y::myY)
    return (myX,myY,foodX,foodY);
}

This code defines the state that the FindFood behavior needs to be concerned with. Together, the state and the reward constructs give the run-time reinforcement learning system all the information it needs to determine how to select the available agent actions in order to achieve the goal, that is, to maximize long-term reward. Note that this is a state abstraction - it contains no elements that are not needed for reasoning about finding food. This is an essential feature of modular behaviors, allowing them to be coded in a truly modular fashion.
Arbitration: Resolving Conflicts Between Subgoals

Once we’ve defined the two adaptive subgoals, we need to define an arbitration function on the enclosing goal, LiveLongProsper. In previous work, we showed that it is impossible to construct an ideal arbitration function automatically [Bhat et al.(2006)], so we cannot employ the compiler to generate an all-purpose arbitration rule. Instead, the programmer must define an arbitration function, either hand-authored or learned.

A hand-authored arbitration function encodes the tradeoffs the programmer believes to be true about the utilities of the subgoals. In this example, we may decide that the benefit of finding food equals the cost of running into a predator; given our reward signals, the arbitrator would select the action maximizing $\frac{1}{10}Q_1(s,a) + Q_2(s,a)$. Alternatively, the hand-authored arbitration function could be independent of the sub-agent Q-values; to simply avoid starvation, for instance, one might consider round-robin scheduling.

Finally, we could try posing LiveLongProsper’s arbitration task as another reinforcement learning task, with its own reward function encapsulating a notion of goodness for living well, as opposed to one that only makes sense for finding food or avoiding a predator. For example, the reward function might provide positive feedback for having more offspring; this would be an “evolutionary” notion of reward.

The reader may wonder why FindFood and AvoidPredator should have their own reward signals if one is available for LiveLongProsper. The reasons should be familiar: modularity and speed of learning. The reward signal for FindFood, for instance, is specifically tailored for the task of finding food, so the learning should converge more quickly than learning via an “indirect” global reward signal. Further, with the right state features, the behavior should be reusable in different contexts. Specifying a reward signal for each behavior allows the reward signals to embody what each behavior truly cares about: FindFood cares about finding grid squares with food, AvoidPredator cares about avoiding the predator, and LiveLongProsper cares about ensuring the future of the species.

---

¹Briefly, arbitration in MRL, as it has been typically defined, can be shown to be equivalent to finding an optimal social choice function and thus falls prey to Arrow’s Impossibility Result. One can avoid this impossibility by having the programmer explicitly define the tradeoffs, essentially repealing the non-dictator property of a “fair” voting system.
Part III

Reference Manual
0.10 Background

A²BL is a multiparadigmatic language incorporating the imperative OOP of Java, reactive planning from AI, and Reinforcement Learning from machine learning. Thus, in order to fully understand A²BL one must have some background in these areas. We assume the reader is familiar with Java and the programming concepts it embodies. In this section, we provide the reader with some basic background knowledge in a few key concepts from Artificial Intelligence (AI). While the presentation here should suffice to understand the remainder of this paper, we provide pointers to more detailed accounts in the literature for the interested reader.

0.10.1 AI Planning

An intelligent agent maximizes goal attainment given available information. In knowledge-based AI, a variety of techniques are used to solve problems. Typical one-step problem-solving scenarios include board games, where an agent must decide on the best move given the current board state. Planning algorithms are used in environments where an agent must find a sequence of actions in order to satisfy its goals. Like most Good Old-Fashioned AI (GOFAI), classical AI planning algorithms rely on deterministic representations; that is, they are not designed to handle probabilistic settings where certain parts of the state space are hidden and some actions don’t always result in exactly the same state change. As we will see in the next sections, machine learning addresses such partially-observable, probabilistic environments directly.

For a more detailed discussion of AI in general, and planning in particular, see [Russell and Norvig(2003)].

0.10.2 Machine Learning

Machine learning algorithms improve their performance on some task as they gain experience. Learning problems specify a task, a performance metric, and a source of training experience. It is important that the training experience provide some feedback so that the learning algorithm can improve its performance. Sometimes the feedback is explicit, as in the case of supervised learning. In supervised learning, an algorithm is presented with a set of examples of a target concept, and the algorithm’s performance is judged by how well it judges new instances of the class. For example, a character recognition system can be trained by presenting it with a large number of examples of the letters of the alphabet, after which it will be able to recognize new examples of alphabetic characters. Some commonly known techniques for such tasks are neural networks, support vector machines, and k-nearest neighbor.
Such learning tasks are said to be \textit{batch-oriented} or \textit{offline} because the training is separate from the performance. In \textit{online} learning, an agent must perform at the same time it is learning, and the feedback here is obtained by acting in the world and succeeding or failing. As we will see in the next section, reinforcement learning algorithms represent this type of algorithm and are particularly well-suited to the construction of intelligent autonomous agents.

For a more detailed discussion of machine learning, see [Mitchell(1997)].

\section*{0.10.3 Reinforcement Learning}

One can think of reinforcement learning (RL) as a machine learning approach to planning. In RL, problems of decision-making by agents interacting with uncertain environments are usually modeled as Markov decision processes (MDPs). In the MDP framework, at each time step the agent senses the state of the environment, and chooses and executes an action from the set of actions available to it in that state. The agent’s action (and perhaps other uncontrolled external events) cause a stochastic change in the state of the environment. The agent receives a (possibly zero) scalar reward from the environment. The agents goal is find a \textit{policy}; that is, to choose actions so as to maximize the expected sum of rewards over some time horizon. An optimal policy is a mapping from states to actions that maximizes the long-term expected reward.

Many RL algorithms have been developed for learning good approximations to an optimal policy from the agent’s experience in its environment. At a high level, most algorithms use this experience to learn value functions (or Q-values) that map state-action pairs to the maximal expected sum of reward that can be achieved by starting from that state-action pair and then following the optimal policy from that point on. The learned value function is used to choose actions. In addition, many RL algorithms use some form of function approximation (parametric representations of complex value functions) both to map state-action features to their values and to map states to distributions over actions (\textit{i.e.}, the policy).

We direct the interested reader to any introductory text on reinforcement learning. There are several such texts, including [Sutton and Barto(1998), Kaelbling et al.(1996)Kaelbling, Littman, and Moore].

\section*{0.10.4 Modular Reinforcement Learning}

Real-world agents (and agents in interesting artificial worlds) must pursue multiple goals in parallel nearly all of the time. Thus, to make real-world partial programming feasible, we must be able to represent the multiple goals of realistic agents and have a learning system that handles them acceptably well in terms of computation time, optimality, and expressiveness. Typically, multiple-goal RL agents are modeled as collections of RL sub-agents that share an
action set. Some arbitration is performed to select the sub-agent action to be performed by
the agent. In contrast to hierarchical reinforcement learning, which decomposes an agent’s
subgoals temporally, we use a formulation of multiple-goal reinforcement learning which
decomposes the agent’s subgoals concurrently. This concurrent decompositional formulation
of multiple-goal reinforcement learning, called modular reinforcement learning (MRL), is
better suited to modeling the multiple concurrent goals that must be pursued by realistic
agents.

Again, readers who want a more thorough explanation of reinforcement learning are referred

0.11 Syntax and Semantics of $A^2$BL

$A^2$BL adds five keywords to the ABL language to enable adaptive programming. For de-
tails of ABL, see [Mateas and Stern(2004)]. The syntax and semantics of $A^2$BL’s five new
keywords are described below.

The adaptive Keyword

The most notable addition in $A^2$BL is the adaptive keyword, used as a modifier for behav-
iors. When modifying a sequential behavior, adaptive signifies that, instead of pursuing the
steps in sequential order, the behavior should learn a policy for which step to pursue, as a
function of the state of the world. Consider the following example:

```plaintext
adaptive sequential behavior FindFood() {
  reward {
    100 if { (FoodWME) }
  }
  state {
    (FoodWME x::foodX y::foodY)
    (SelfWME x::myX y::myY)
    return (myX,myY,foodX,foodY);
  }
  subgoal MoveNorth();
  subgoal MoveSouth();
  subgoal MoveEast();
  subgoal MoveWest();
}
```

The adaptive modifier on this behavior tells the $A^2$BL run-time system to learn how to
sequence the subgoals specified within the behavior as it interacts in the environment. The
programmer codes a partial specification of the problem—the subgoals—and the system
learns the rest, namely, how to sequence them optimally in a dynamic environment.
The state Construct

As there could be a large amount of information in working memory (which is the agent’s perception of the state of the world), we have introduced a state construct to allow the programmer to specify which parts of working memory the behavior should pay attention to in order to learn an effective policy. This allows for human-authored state abstraction, a fundamental concept in reinforcement learning. In the example in the previous section, the state was specified as:

```latex
state {
  (FoodWME x::foodX y::foodY)
  (SelfWME x::myX y::myY)
  return (myX,myY,foodX,foodY);
}
```

This tells the A2BL runtime system what comprises the state to be used in its RL algorithms for this particular behavior or task. The policy learned for food-finding will be predicated on this state. Note that the state contains no elements that are not needed for reasoning about finding food. This is an essential feature of modular behaviors, allowing them to be coded in a truly modular fashion.

The success_condition Condition

In ABL, a behavior normally succeeds when all its steps succeed. Because it is unknown which steps the policy will ultimately execute, adaptive behaviors introduce a new continually-monitored condition, the success_condition, which encodes the behavior’s goal. When the success condition becomes true, the behavior immediately succeeds. In our example agent, there is no such end-state goal. The agent must continually find food and avoid the predator.

The reward Construct

To learn a policy at all, the behavior needs a reinforcement signal. With the reward construct, authors specify a function that maps world state to such a reinforcement signal. In natural analogy to existing ABL constructs, these new constructs each make use of WME tests for reasoning and computing over working memory. Consider the following code:

```latex
reward {
  100 if { (FoodWME) }
}
```

The code above says that, if the agent finds the food, it gets a large positive reward (recall that WMEs are the mechanism by which an agent senses the world in ABL and in A2BL).
This reward is used by the RL algorithms to learn an action selection policy that maximizes long-term reward.

collection Behaviors

An adaptive collection behavior is specifically designed for modeling the concurrency of MRL. This type of behavior contains within it several adaptive sequential behaviors, which correspond to the sub-agents in the MRL framework. Consider the following code:

```java
adaptive collection behavior LiveLongProsper() {
    subgoal FindFood();
    subgoal AvoidPredator();
}
```

This code defines the LiveLongProsper behavior as consisting of two concurrent subgoals – FindFood and AvoidPredator. A²BL will attempt to pursue both of the goals concurrently while the agent is running.

0.12 Error Handling

A²BL uses Java’s Exception system for error handling. However, where the state of a normal sequential program is defined by a stack, the state of an ABL/A²BL program is defined by a tree, the Active Behavior Tree (ABT) in particular. ABL includes a special debugging tool to enable programmers to see the ABT for ABL programs.

0.13 A BNF Grammar for A²BL

While A²BL is being written using ANTLR, ABL uses JavaCC. The following grammar is generated by JJDoc, with additional elements for A²BL. Any productions referred to but not defined are part of the Java grammar itself, and are mostly self-explanatory.

```plaintext
BehaviorUnit ::= ( "package" Name ";" )? ( ( ImportDeclaration )
| ( ConstantDeclaration ) )* <BEHAVING_ENTITY> AblName <LBRACE>
( TeamNeededForSuccessDefaultDeclaration )?
( DecisionCycleSMCallDeclaration )? ( ConflictDeclaration )* ( AblDeclaration )* ( BehaviorDefinition )* ( InitialTree )* <RBRACE> <EOF>

ConstantDeclaration ::= <CONSTANTS> Name ( "." "*" )? ";"
ConflictDeclaration ::= <CONFLICT> AblName ( AblName )+ <SEMICOLON>
PropertyDeclaration ::= <PROPERTY> Type AblName <SEMICOLON>
```
AblDeclaration ::= \((\text{AblVariableDeclaration} \ \text{SEMICOLON})\) |
WMERegistration |
ActionRegistration |
WMEDeclaration |
PropertyDeclaration

WMERegistration ::= \(<\text{REGISTER}>\) \<\text{WME}>\ \text{WMEclass} \ \text{WITH} \ \text{Name} \ \text{SEMICOLON}>

ActionRegistration ::= \(<\text{REGISTER}>\) \<\text{ACTION}>\ \text{AblName} \ <\text{LPAREN}>(\ \text{AblSimpleType} \ <\text{COMMA}>=\ \text{AblSimpleType})*? \ <\text{RPAREN}>\ <\text{WITH}>\ \text{Name} \ \text{SEMICOLON}>

BehaviorDefinition ::= \((\text{BehaviorTypeModifier})^*\) \text{BehaviorType} \<\text{BEHAVIOR}>\ \text{AblName} \ <\text{LPAREN}>(\ \text{AblSimpleType} \ \text{AblSimpleType})*? \ <\text{RPAREN}>\ <\text{LBRAbrace}>(\ \text{BehaviorModifiers})^*\<\text{RBRAbrace}>\text{BehaviorStep})*\ <\text{RBRAbrace}>

BehaviorModifiers ::= \((\text{Precondition} \ | \ \text{Specificity} \ | \ \text{ContextCondition} \ | \ \text{EntryCondition} \ | \ \text{NumberNeededForSuccess} \ | \ \text{TeamMemberSpecifier} \ | \ \text{SuccessCondition} \ | \ \text{ReinforcementSignals} \ | \ \text{ReinforcementState})\)

Specificity ::= \(<\text{SPECIFICITY}>\) \<\text{INTEGER_LITERAL}>\ \text{SEMICOLON}>

NumberNeededForSuccess ::= \(<\text{NUMBER_NEEDED_FOR_SUCCESS}>\) \<\text{INTEGER_LITERAL}>\ \text{SEMICOLON}>

TeamMemberSpecifier ::= \(<\text{TEAMMEMBERS}>\) \<\text{IDENTIFIER}>\)+\ <\text{SEMICOLON}>

InitialTree ::= \(<\text{INITIAL_TREE}>\) \<\text{LBRAbrace}>(\ \text{BehaviorStep})^*\ <\text{RBRAbrace}>

BehaviorType ::= \((\text{<SEQUENTIAL} \ | \ \text{PARALLEL} \ | \ \text{COLLECTION})\)

BehaviorTypeModifier ::= \((\text{<ATOMIC} \ | \ \text{<JOINT} \ | \ \text{<ADAPTIVE}\))

Precondition ::= \(<\text{PRECONDITION}>\) \<\text{LBRAbrace}>(\ \text{TestExpression}) \<\text{RBRAbrace}>

TestExpression ::= \((\ \text{IDENTIFIER})*?\) \ <\text{WMETestSequence}>\ +\ <\text{IDENTIFIER}>\ \text{WMETestSequence}

DefaultMemoryLookahead ::= \text{<IDENTIFIER>\ WMETestSequence}

WMETestSequence ::= \(<\text{LBRAbrace}>\) \text{<IDENTIFIER>\ (<\text{WMETest} \ | \ <\text{LPAREN}>\ \text{ConditionalExpression} \ <\text{RPAREN}>)* \ <\text{RBRAbrace}>\text{WMETest} \ (<\text{LPAREN} \text{ConditionalExpression} \ <\text{RPAREN}>)^*\ <\text{RBRAbrace}>

WMETest ::= \((\text{<BANG}>)?\) \ <\text{Name}> \ <\text{ASSIGN}>)?\ <\text{LPAREN}>\ \text{WMEclass} \ (<\text{WMEFieldTest})* \ <\text{RPAREN}>

WMEClass ::= \text{AblName}

WMEFieldTest ::= \text{<IDENTIFIER>\ (<V_BIND} \ | \ <GT} \ | \ <LT} \ | \ <EQ} \ | \ <LE} \ | \ <GE} \ | \ <NE} \ <\text{AblExpression}>

AblLiteral ::= \((\text{<INTEGER_LITERAL} \ | \ <FLOATING_POINT_LITERAL} \ | \ <DOUBLE_LITERAL} \ | \ <CHARACTER_LITERAL} \ | \ <STRING_LITERAL} \ | \ <BOOLEAN_LITERAL} \ | \ <NULL_LITERAL})\ <\text{AblExpression}>

AblExpression ::= \((\ \text{Name} \ | \ \text{AblLiteral})\)

ContextCondition ::= \(<\text{CONTEXT_CONDITION}>\) \<\text{LBRAbrace}>(\ \text{TestExpression}) \<\text{RBRAbrace}>

EntryCondition ::= \(<\text{ENTRY_CONDITION}>\) \<\text{LBRAbrace}>(\ \text{TestExpression}) \<\text{RBRAbrace}>

28
SuccessCondition ::= <SUCCESS_CONDITION> <LBRACE> TestExpression <RBRACE>

ReinforcementSignals ::= <R_SIGNAL> <LBRACE> ( ReinforcementSignal )* <RBRACE>

ReinforcementSignal ::= AblExpression <IF> <LBRACE> TestExpression <RBRACE>

ReinforcementState ::= <STATE> <LBRACE> StateCondition ( ( AblVariableDeclaration <SEMICOLON> )
| ( Statement ) )* StateReturn <RBRACE>

StateCondition ::= <LBRACE> TestExpression <RBRACE>

StateReturn ::= <STATE> <RETURN> <LPAREN> AblExpression ( <COMMA> AblExpression )* <RPAREN> <SEMICOLON>

BehaviorStep ::= ( ( <WITH> <LPAREN> StepModifier ( <COMMA> StepModifier )* <RPAREN> )? ModifiableStep |
| FailStep |
| SucceedStep |

ModifiableStep ::= PrimitiveActStep |
| MentalActStep |
| GoalStep |
| ModifyStep |
| AnonymousStep |
| WaitStep |

PrimitiveActStep ::= PrimitiveAct

MentalActStep ::= MentalAct

GoalStep ::= Goal

WaitStep ::= Wait

AnonymousStep ::= AnonymousBlock

ModifyStep ::= ( ( <MOVE_STEP> Name INTEGER_LITERAL Name INTEGER_LITERAL Name Name <SEMICOLON> )
| ( <DELETE_STEP> Name INTEGER_LITERAL Name Name <SEMICOLON> )
| ( <ADD_STEP> Name INTEGER_LITERAL Name Name BehaviorStep ) )

FailStep ::= <FAIL> <SEMICOLON>

SucceedStep ::= <SUCCEED> <SEMICOLON>

AnonymousBlock ::= BehaviorType <LBRACE> ( AblVariableDeclaration <SEMICOLON> )* ( BehaviorStep )* <RBRACE>

StepModifier ::= SuccessTest |
| PriorityModifier |
| IGNORE_FAILURE |
| Persistence |
| EFFECT_ONLY |
| TEAM_EFFECT_ONLY |
| NamedProperty |
| <POST> |
| <POST_TO> IDENTIFIER |
| <ONE_NEEDED_FOR_SUCCESS |
| TEAM_NEEDED_FOR_SUCCESS |

SuccessTest ::= <SUCCESS_TEST> <LBRACE> TestExpression <RBRACE>

PriorityModifier ::= ( ( <PRIORITY_MODIFIER> INTEGER_LITERAL )
| ( <PRIORITY> INTEGER_LITERAL )

Persistence ::= ( ( <PERSISTENT> PersistenceType )
| ( <PERSISTENT> )

NamedProperty ::=   <PROPERTY> AblName AblExpression
PersistenceType ::= ( <WHENFAILS> | <WHENSUCCEEDS> )

PrimitiveAct ::= <ACT> AblName LPAREN ( AblExpression ( <COMMA> AblExpression )* )? RPAREN SEMICOLON

MentalAct ::= <MENTAL_ACT> LBRACE ( ( AblVariableDeclaration SEMICOLON ) | ( Statement ) )* RBRACE

Goal ::= ( <JOINT> )? ( <SUBGOAL> | <SPAWNGOAL> ) AblName LPAREN ( AblExpression ( <COMMA> AblExpression )* )? RPAREN ( <AT> Name )? SEMICOLON

Wait ::= <WAIT> SEMICOLON

AblVariableDeclaration ::= Type VariableDeclarator ( "", VariableDeclarator )*  

AblSimpleType ::= ( PrimitiveType | <IDENTIFIER> )

AblName ::= <IDENTIFIER>

WMEDeclaration ::= <WME> AblName ( <EXTENDS> AblName )* LBRACE ( AblVariableDeclaration SEMICOLON )* RBRACE

TeamNeededForSuccessDefaultDeclaration ::= <JOINT_GOAL_SUCCESS_NEGOTIATION> ( <TEAM_NEEDED_FOR_SUCCESS> | <ONE_NEEDED_FOR_SUCCESS> ) SEMICOLON

DecisionCycleSMCallDeclaration ::= <DECISION_CYCLE_SM_CALL> Name SEMICOLON
Part IV

Appendix
ANTLR Grammar for A²BL

The ANTLR grammar for A²BL is given below. Note that the grammar inherits the Java ANTLR grammar. A²BL programs are translated to Java programs, and arbitrary Java code can be inserted into certain parts of A²BL programs.

```java
header {
    package abl.compiler;

    import java.util.*;
}

/**
 * This class is responsible only for constructing the AST.
 */
class AblParser extends JavaRecognizer;

options {
    exportVocab=Temp;
}
tokens { // some virtual token types
    WME_DECL; TEST_EXPR; TEST_SEQUENCE; WME_TEST; ANON_BLOCK;
}

private JavaBuilder builder;
void setBuilder(JavaBuilder b) { builder = b; }

// copied from java15.g, needed by JavaBuilder
/**
 * Counts the number of LT seen in the typeArguments production.
 * It is used in semantic predicates to ensure we have seen enough closing '>' characters; which actually may have been either GT, SR or BSR tokens.
 */
private int ltCounter = 0;

// Rules from the java15.g Java grammar used by this Abl grammar:
// annotations, packageDefinition, importDefinition, typeSpec,
// conditionalExpression, constant, compoundStatement, expression,
// declaration, builtInType, identifier, argList, expressionList,
// parameterDeclarationList, & all the tokens except V_BIND ("::")
// and literals in this file

// Invariants: a child AST must be fully prepared before added to a parent AST
// A reference to the parent AST should be established ASAP so the children of the child AST can access its grandparent AST if necessary (e.g. Steps may need to access the Entity).

behaviorUnit
(EntityAST entity = (EntityAST)#[LITERAL_behaving_entity,
    "behaving_entity","abl.compiler.EntityAST"];)
: (p:packageDefinition
    {entity.setPackageName(builder.identifier(#p.getFirstChild().getNextSibling()).toString());} )?
(i:importDefinition
    {entity.addImport(builder.identifierStar(#i.getFirstChild()).toString());} )*
(c:constants
    {entity.addConstant(builder.identifier(#c.getFirstChild()).toString());} )*"behaving_entity"!
    name:IDENT (entity.setName(#name.getText()));)
LCURLY!
( teamNeededForSuccessDefaultDeclaration[entity]
    | decisionCycleSMCallDeclaration[entity]
    | registration[entity]
```

32
behaviorDefinition[entity]
| initialTree[entity] )* 
RCURLY!
EOF!

{## = #(entity,##);}

};

costants

: "constants" identifier SEMI!
;

teamNeededForSuccessDefaultDeclaration[EntityAST entity]
: "joint_goal_success_negotiation"

| "team_needed_for_success" {entity.setTeamNeededForSuccess(true);} 
| "one_needed_for_success" {entity.setTeamNeededForSuccess(false);} 
) SEMI!
;

decisionCycleSMCallDeclaration[EntityAST entity]
{String id;}
: "decision_cycle_sm_call" i:identifier SEMI!

{entity.setDecisionCycleSMCall(builder.identifier(#i).toString());}
;

registration[EntityAST entity]
{List conflict = new LinkedList();
String superName = null;}
: declaration SEMI!

{entity.addDeclaration();}
| "conflict" a:IDENT 
{conflict.add(#a.getText());}
(b:IDENT {conflict.add(#b.getText());})+ SEMI!

{entity.addConflict(conflict);}
| "register"!
{wme" vmeClass:IDENT "with"! i1:identifier SEMI!
{entity.addWme(#vmeClass.getText(),builder.identifier(#i1).toString());}
| "register"!
{action"|"act"" actName:IDENT LPAREN! typeList RPAREN!
"with"! i2:identifier SEMI!
{entity.addAct(#actName.getText(),builder.identifier(#i2).toString()); ##.setType(LITERAL_act);}
| "property" propType:typeSpec[true] propName:IDENT SEMI!
{entity.addProperty(#propType,#propName.getText());}
| "wme" wmeName:IDENT ("extends"! i:IDENT 
{superName=#i.getText();})? LCURLY! (declaration SEMI!)* RCURLY!

{entity.addWmeDecl(#wmeName.getText(),superName); ##.setType(WME_DECL);}
;

typeList // possibly empty comma-separated list of types
: (typeSpec[true] (COMMA! typeSpec[true])*);
;

behaviorDefinition[EntityAST entity]
{BehaviorAST behavior = (BehaviorAST)#[LITERAL_behavior,
"behavior","abl.compiler.BehaviorAST"];}

: {

behavior.setEntity(entity); // otw NullPointerException when adding steps
behavior.setId(entity.getUniqueBehaviorId());

} (behaviorTypeModifier[behavior])* 
behaviorType[behavior]
"behavior"! name:IDENT ( behavior.setName(#name.getText()); }
LPAREN! parameterDeclarationList RPAREN!
LCURLY! (behaviorModifier[behavior])* (declaration SEMI!)*

behaviorStep[behavior])* RCURLY!

{ ## = #(declaration SEMI!)

};

behaviorTypeModifier[BehaviorAST behavior]
: "atomic" (behavior.setAtomic());
| "joint" {behavior.setJoint();}
"adaptive" {behavior.setAdaptive();}

behaviorType[BehaviorAST behavior] // FIXME remove magic numbers
  "sequential" {behavior.setBehaviorType(0);}
  "parallel" {behavior.setBehaviorType(1);}
  "collection" {behavior.setBehaviorType(2);}

behaviorModifier[BehaviorAST behavior]
  {int num = 0;
   List list = new LinkedList();
   "specificity"^ num=integer SEMI! {behavior.setSpecificity(num);}
   "teammembers"^ (i:IDENT{list.add(#i.getText());})+ SEMI! {behavior.setTeamMembers(list);}
   "number_needed_for_success"^ num=integer SEMI! {behavior.setNumberNeededForSuccess(num);}
   "precondition"" LCURLY! ti:testExpression RCURLY! {behavior.setPrecondition(#t1);}
   "entry_condition"" LCURLY! t2:testExpression RCURLY! {behavior.setEntryCondition(#t2);}
   "context_condition"" LCURLY! t3:testExpression RCURLY! {behavior.setContextCondition(#t3);}
   "success_condition"" LCURLY! t4:testExpression RCURLY! {behavior.setSuccessCondition(#t4);}
   "reward"" LCURLY! (expression "if"! LCURLY! testExpression RCURLY!)* RCURLY! {behavior.setRewardDefinition();}
   "state"" LCURLY! LCURLY! testExpression RCURLY! ( statement )*
   "state"! "return"! LPAREN! expressionList RPAREN! SEMI! RCURLY! {behavior.setStateDefinition();}
  ;

behaviorStep[BehaviorAST behavior]
  {StepAST step = (StepAST)#[LITERAL_with,"","abl.compiler.StepAST"];
   BehaviorAST anon = (BehaviorAST)#[ANON_BLOCK,"","abl.compiler.BehaviorAST"];
   boolean hasModifiers = false;
   (goalStep[null]) => goalStep[behavior]
   |
   goalStep[null] => goalStep[behavior]
  ;
("with"! LPAREN! stepModifier[step]
  (COMMA! stepModifier[step])* RPAREN! (hasModifiers = true;))?}
| "act"! IDENT LPAREN! argList RPAREN! SEMI!
  {step.setType(LITERAL_act);}
| "mental_act"! compoundStatement
  {step.setType(LITERAL_mental_act);}
| "move_step"! "from"! integer ("before"|"after") integer "in"! identifier SEMI!
  {step.setType(LITERAL_move_step);}
| "delete_step"! "from"! integer "in"! identifier SEMI!
  {step.setType(LITERAL_delete_step);}
| "add_step"! ("before"|"after") integer "in"! identifier behaviorStep[null]
  {step.setType(LITERAL_add_step);}
| "wait"! SEMI!
  {step.setType(LITERAL_wait); if(!hasModifiers) step.setId(-3); } // FIXME remove magic numbers }
| "fail_step"! SEMI! {step.setType(LITERAL_fail_step); step.setId(-2); }
| "succeed_step"! SEMI! {step.setType(LITERAL_succeed_step); step.setId(-1); }
|} = #(step,##);
step.setBehavior(behavior);
if (step.getId() == null) step.setId(behavior.getUniqueStepId());
behavior.addStep(step);
| | behaviorType[anon] LCURLY! (declaration SEMI!)* (behaviorStep[anon])+ RCURLY!
  { ## = #(anon,##); behavior.getEntity().addBehavior(anon); /* FIXME addStep() here? */ } ;
goalStep[BehaviorAST behavior]
{GoalStepAST step = GoalStepAST#LITERAL_with,"abl.compiler.GoalStepAST";}
: ("with"! LPAREN! goalStepModifier[step]
  (COMMA! goalStepModifier[step])* RPAREN!)?
  ("subgoal"! IDENT LPAREN! argList RPAREN! SEMI!
  "spawngoal"! IDENT LPAREN! argList RPAREN! ("at"! identifier)? SEMI!
  {step.setType(LITERAL_subgoal);}
  {step.setType(LITERAL_spawngoal);})
|} = #(step,##);
step.setBehavior(behavior);
step.setId(behavior.getUniqueStepId());
behavior.addStep(step);
|;  
stepModifier[StepAST step]
{int n=0;}
: "success_test"^ LCURLY! t:testExpression RCURLY!
  {step.setSuccessTest(#t);}
| "property"^ IDENT expression
| "priority_modifier"^ n=integer {step.setPriorityModifier(n);}
| "priority"^ n=integer {step.setPriority(n);}
| "ignore_failure" {step.setIgnoreFailure();}
| "persistent"^ 
  { "when_fails" {step.setPersistentWhenFails();}
  | "when_succeeds" {step.setPersistentWhenSucceeds();}
  | /* nothing */ {step.setPersistent();}
  } }
| "effect_only" {step.setEffectOnly();}
| "team_effect_only" {step.setTeamEffectOnly();}
| "post" {step.setPost();}
| "post_to"^ memory:IDENT {step.setPostTo(#memory.getText());}
  ;
goalStepModifier[GoalStepAST step]
: stepModifier[step]
  | "one_needed_for_success" {step.setTeamNeededForSuccess(false);}

| "team_needed_for_success" {step.setTeamNeededForSuccess(true);}
|
initialTree[EntityAST entity]
: behavior:"initial_tree"<AST=abl.compiler.BehaviorAST>
{
    #behavior.setEntity(entity); // otw NullPointerException when adding steps
    #behavior.setId(entity.getUniqueBehaviorId());
    #behavior.setBehaviorType(2); // CollectionBehavior FIXME remove magic number
    #behavior.setName(entity.getName() + "_RootCollectionBehavior");
    #behavior.setSignature(entity.getName() + "_RootCollectionBehavior()");
}
LCURLY! (behaviorStep[#behavior])+ RCURLY!
{
    entity.setInitialTree(#behavior);
}
|
// convenience method

integer returns [int val=0]
{int sign=1;}
: (MINUS {sign=-1;})? i:NUM_INT {val=sign*Integer.parseInt(#i.getText());}
|
class AblLexer extends JavaLexer;

options {
importVocab=Temp;
exportVocab=Abl;
}

V_BIND : "::" ;
Bibliography


