The Pencil Compiler Architecture

Christopher Conway Cheng-Hong Li Megan Pengelly

3 December 2002

1 Architecture

The Pencil compiler consists of several major blocks which are common in compiler designs: lexer, parser, symbol table, error handler, the runtime environment, and the code generator. The relationship between these components is demonstrated in Figure 1. The code generator is not explicitly shown; but with the parser these two components are implemented using a Director and Builder design pattern. The input to the compiler are Pencil specification files (which have, by convention, the suffix .pen) and the final output from the compiler is translated Java code. The compiler takes one pen file once a time, and translates it to Java. The lexer is implemented in JFlex, a version of lex for Java. The parser is generated by CUP, a version of yacc for Java. The error handler is deals with error recovery when syntax errors are encountered. It informs the code generator not to generate the final output whenever a syntax error is found, and prompts the user with an appropriate error message. The runtime system is a set of predefined classes that are inherited by the generated Java classes. The runtime classes provide the basic computation necessary for Petri net simulation and execution. Between the parser and the code generator there is a clear interface making these two components highly modular. The Builder pattern allows the change of front-end and back-end design not to affect the other part of the design if the interface between them is followed.

Figure 1: The block diagram of PENCIL compiler.

The entry point of the compiler is the class pencil.Main. The main() method parses the command line argument, and initializes the parser to begin processing the Pencil source file. The interface between the lexer and the parser is the lexer’s next_token() method. The parser’s task is the “director” of the
code generator, which is the “builder” of the target code. Semantic actions on parser productions invoke a set of builder method, constructing an abstract Petri net representation from the source code. If a syntax error occurs, the parser invokes the error handler (SynErrHandler). The error handler prints an appropriate error message, and also informs the builder that a syntax error has occurred. When parsing is finished, the build() method of the Builder class is invoked, and the target code is generated.

The Builder pattern most suits a compiler with different targets. The separation of a Director and a Builder not only allows the creation of different targets with the same front-end components (the parser, lexer, error handler, and symbol table), but also minimizes the effort to improve both the front-end and the back-end, and add on more language features. The parser is the Director, and it directs the concrete instances of NetBuilder to build the target, with a sequence of calls. The Java interface NetBuilder defines the interface between the parser and the concrete builders. A Builder class implements this interface and interprets the sequence of calls from the Director as instructions for building a Petri net class file. The diagram 2 demonstrates the relationship between the parser (Director) and the NetBuilder (Builder).

```
Parser p = new Parser(new Lexer(in)) ;
// ...

NetBuilder b = null ;
```
if( debug ) {
    b = new DebugBuilder();
} else if (textSim){
    b = new SimBuilder();
} else{
    b = new DefaultBuilder();
}
p.setBuilder(b);

The interface between the parser and the NetBuilder defines an abstract model of Petri nets. The capture of the abstract model of Petri nets is crucial both to the Director and the Builder. Language extensions and improved back-end components can be introduced without breaking the Director-Builder interface. The interface consists mainly of methods to inform the Builder of place definitions, transition definitions, transition firing types and arcs between places and transitions. More specifically, the following methods are used to set the place definitions:

public void startPlace(String name) ;
public void setMarking(String m) ;
public void setBound(int b) ;
public void setBound(String b) ;
public void endPlace() ;

The startPlace() is called when the parser finds the beginning of a place definition in the source code. The parser then communicates the initial marking and the upper-bound (on the number of tokens this place can have most) information by calling setMarking and setBound. Finally the endPlace() method is used to signal the end of a place definition.

The transition-related methods are defined in a similar way. They are:

public void startTransition(String name) ;
public void addTransition(String name) ;
public void setReturn(int r) ;
// ....
public void setFire(Integer f) ;
public void setJava(String code) ;
public void endTransition() ;

The sequence diagram in Figure 3 illustrates the flow of calls between pencil.Main, the parser, and NetBuilder.

2 The Runtime Environment

The runtime environment provides the basic infrastructure of the operation of the generated Java files. A set of classes model abstract Petri nets, transi-
tions, arcs, and places. The generated Java files are composed of classes inheriting these runtime classes. These classes are class PetriNet, Transition, Place, and Arc. The main body of the generated Java class is a subclass of class PetriNet, and in its constructor it instantiates a concrete subclass of Transition. The concrete subclass of Transition implements the associated Java action from its definition in the Pencil source code. The following generated Java code fragment shows how they work:

class HelloWorld extends pencil.runtime.PetriNet {

    public HelloWorld() {
        pencil.runtime.Place p2 = new pencil.runtime.Place() ;
        add(p2) ;

        pencil.runtime.Place p1 = new pencil.runtime.Place(1) ;
        add(p1) ;

        pencil.runtime.Transition t1 = new pencil.runtime.Transition() {
            public Object onFire() {
                System.out.println("Hello, world!");
                return null ;
            }
        }
    }
}
3 Error Recovery

The error handling and recovery of Pencil adopts two kinds of techniques to recover from parsing errors. One is the error production, and the other one is the panic mode. Error productions are a set of grammar rules that generate predictable incorrect inputs from users. They are very useful for catching common minor syntax errors. Any reduction of these error productions means some syntax error has been recovered. The panic mode is implemented the support of CUP's special ‘error’ token. This special token represents an error construct in the source file, and the parser will re-synchronize the parsing under the condition that a sufficient amount (default value is 3) of correct shifts follows the last error input. Panic mode is useful for bigger syntax errors. The use of the “error” tokens requires great care, usually with the help of the generated parsing table.

The following error production rule tries to capture the missing semicolon error, which is a common error in the input pen files:

PlaceDecl ::= 
    PLACE PlaceList SEMI 
// ....

    /* Error Productions */
    // catch missing semicolon.
    | PLACE PlaceList:err
    {: parser.errHandler.onErrProduction(SynErrHandler.ERR_MISSING_SEMI, 
            errleft, errright);
        parser.synErr = true;
    };

The panic mode technique is for other non-predictable parsing errors. The placement of the “error” tokens are not quite straightforward. The following fragment of the production demonstrates the use of the “error” token:

PlaceDecl ::= 
    PLACE PlaceList SEMI
    | PLACE Place PlaceDef
      /* Panic Mode */
    | PLACE error:err
    {: parser.errHandler.onError(SynErrHandler.ERR_PLACE_MISSING_COMMA_SEMI, 
            errleft, errright);
        parser.synErr = true;
Here the "error" token represents an incorrect PlaceList construct.