The Design and Implementation of SPARK, a Toolkit for Implementing Domain-Specific Languages

John Aycock

Department of Computer Science, University of Calgary, Alberta, Canada

SPARK is a toolkit for implementing domain-specific languages in Python. It is somewhat unusual in that its intended audience includes users who do not necessarily have a background in compilation; this choice impacts both the design and implementation of SPARK. We introduce SPARK in this paper and discuss major issues that have arisen in its design and ongoing development.

Keywords: compiler tools, domain-specific languages, software design

1. Introduction

SPARK, the Scanning, Parsing, And Rewriting Toolkit, is a toolkit which assists in the implementation of domain-specific languages in Python. First unveiled in 1998, SPARK is now on its sixth release. In that time, SPARK has been used by ourselves and others to implement a number of domain-specific languages; a selection of these projects are listed below.

- Compiling Guide [23], a web programming language
- Bytecode decompilation
- GUI building
- Linux[1] kernel configuration system [26]
- Interfacing with IRAF (astronomical software) [33]
- Fortran interface description [9]

Early on, we decided to target SPARK to a specific group of users, a somewhat unusual choice in that it included users without a background in compilation. In the remainder of this paper we introduce SPARK, then discuss our design choices in more detail and how they have influenced SPARK's implementation.

2. SPARK[2]

We begin with a simple model of a compiler having only four phases, as shown in Figure 1:

1. Scanning, or lexical analysis, breaks the input stream into a list of tokens and their attributes (if any).
2. Parsing, or syntax analysis, resulting in the construction of an abstract syntax tree (AST). In this section, we use a typical expression grammar as our running example:

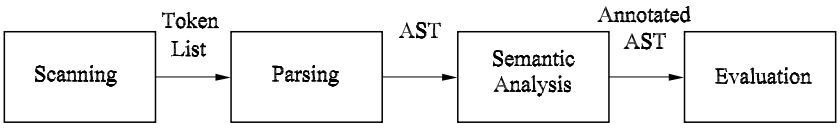

Fig. 1. Compiler model. Note the absence of feedback edges.

[1] Linux is a trademark of Linus Torvalds.
[2] A very early version of the work in this section appeared in [2].

```
expr ::= expr + term
expr ::= term
term ::= term * factor
term ::= factor
factor ::= number
```

3. Semantic analysis, performed by traversing the AST one or more times.
4. Evaluation, which may directly interpret the program, or output code in C or assembly which would implement the input program. Evaluation may be implemented by another traversal of the AST, or by matching patterns in the AST.

Each phase performs a well-defined task, and passes a data structure on to the next phase; Grune et al. [15] refer to this as a "broad compiler." Note that information only flows one way, and that each phase runs to completion before the next one starts. This is in contrast to oft-used techniques based on a symbiosis between scanning and parsing, where several phases may be working concurrently, and a later phase may send some feedback to modify the operation of an earlier phase.

Certainly not all domain-specific language compilers will fit this model, but it is extremely clean and elegant for those that do. The main function of the compiler, for instance, distills into three lines of Python code which reflect the compiler's structure:

```
f = open(filename)
evaluate(semantic(parse(scan(f))))
f.close()
```

How are these phases implemented using SPARK? A common theme throughout SPARK is that the user should have to do as little work as possible. For each phase, SPARK supplies a class which performs most of the work; these are summarized in Table 1. The user's job is simply to create subclasses which customize SPARK's base classes.

Phase	*Class*
Lexical analysis	GenericScanner
Syntax analysis	GenericParser
Syntax analysis	GenericASTBuilder
Semantic analysis	GenericASTTraversal
Evaluation	GenericASTTraversal
Evaluation	GenericASTMatcher

Table 1. SPARK classes, by functionality. Some classes are potentially useful for more than one phase.

2.1. Lexical Analysis

A user creates a scanner by creating a subclass of GenericScanner, SPARK's generic scanner class. In this subclass, the user specifies the regular expressions that the scanner should look for, and actions (consisting of arbitrary Python code) to be performed based on the type of token found.

Below is a simple scanner to tokenize expressions for our running example. The parameter to the action routines is a string containing the part of the input that was matched by the regular expression.

```
class SimpleScanner(GenericScanner):
    def __init__(self):
        GenericScanner.__init__(self)

    def tokenize(self, input):
        self.rv = []
        GenericScanner.tokenize(self, input)
        return self.rv

    def t_whitespace(self, s):
        r' \s+ '

    def t_op(self, s):
        r' \+ | \* '
        self.rv.append(Token(type=s))

    def t_number(self, s):
        r' \d+ '
        t = Token(type='number', attr=s)
        self.rv.append(t)
```

A few words about the syntax and semantics of Python are in order. This code defines the class `SimpleScanner`, a subclass of `GenericScanner`. All methods have an explicit `self` parameter; `__init__` is the class' constructor, and it is responsible for invoking its superclass's constructor if necessary. Methods may optionally begin with a documentation string which is ignored by Python's interpreter but, unlike a regular comment, is retained and accessible at run-time. For example, the documentation string for the `t_number` method is `r'\d+'`, where the prefix "`r`" denotes a "raw" string in which backslash characters are not treated as escape sequences. A method which is empty (save for an optional documentation string) has no effect when executed.

Object instantiation uses the same syntax as function calls; in the above code, `Token` objects are being created. Both object instantiation and function invocation can make use of "keyword arguments," which permit actual and formal parameters to be associated by name rather than by their position in the argument list. Above, `type=s` passes `s` to the `type` parameter. Finally, `[]` denotes the empty list.

Returning to the scanner itself, each method whose name begins with "`t_`" is an action; the regular expression for the action is placed in the method's documentation string. (The reason for this unusual design, using reflection, is explained in Section 3.2.1.)

When the tokenized method is called, a list of `Token` instances is returned, one for each operator and number found. The code for the `Token` class is omitted; it is a simple container class with a type and an optional attribute. White space is skipped by `SimpleScanner`, since its action code does nothing. Any unrecognized characters in the input are matched by a default pattern, declared in the action `GenericScanner.t_default`. This default method can of course be overridden in a subclass.

Scanners made with GenericScanner are extensible, meaning that new tokens may be recognized simply by subclassing. To extend `SimpleScanner` to recognize floating-point number tokens is easy:

```
class FloatScanner(SimpleScanner):
    def __init__(self):
        SimpleScanner.__init__(self)

    def t_float(self, s):
        r' \d+ \. \d+ '
        t = Token(type='float', attr=s)
        self.rv.append(t)
```

How are these classes used? Typically, all that is needed is to read in the input program, and pass it to an instance of the scanner:

```
def scan(f):
    input = f.read()
    scanner = FloatScanner()
    return scanner.tokenize(input)
```

Here, the entire input is read at once with the `read` method. Once the scanner is done, its result is sent to the parser for syntax analysis.

2.2. Syntax Analysis

The outward appearance of GenericParser, our generic parser class, is similar to that of GenericScanner.

A user starts by creating a subclass of GenericParser, containing special methods which are named with the prefix "`p_`". These special methods encode grammar rules in their documentation strings; when a grammar rule is recognized by GenericParser, the rule's associated "action" code in the method is executed.

The expression parser subclass is shown below. Here, the actions are building the AST for the input program from the bottom up. `AST` is also a simple container class; each instance of `AST` corresponds to a node in the tree, with a node type and possibly child nodes. The grammar's start symbol is passed to the constructor. In the code, `ExprParser`'s constructor assigns a default value to its start symbol argument so that it may be changed later by a subclass.

```
class ExprParser(GenericParser):
    def __init__(self, start='expr'):
        GenericParser.__init__(self, start)

    def p_expr_1(self, args):
        ' expr ::= expr + term '
        return AST(type=args[1], left=args[0],
                                 right=args[2])

    def p_expr_2(self, args):
        ' expr ::= term '
        return args[0]

    def p_term_1(self, args):
        ' term ::= term * factor '
        return AST(type=args[1], left=args[0],
                                 right=args[2])

    def p_term_2(self, args):
        ' term ::= factor '
        return args[0]

    def p_factor_1(self, args):
        ' factor ::= number '
        return AST(type=args[0])

    def p_factor_2(self, args):
        ' factor ::= float '
        return AST(type=args[0])
```

The "`args`" passed in to the actions are based on a similar idea used by Yacc [18, 22]. Each terminal or nonterminal symbol on a rule's right-hand side has an attribute associated with it. For token symbols like `+`, this attribute is the token itself. All other symbols' attributes come from the return values of actions which, in the above code, means that they are subtrees of the AST.

The index into `args` corresponds to the position of the symbol in the rule's right-hand side. In the running example, the call to `p_expr_1` has `len(args) == 3`: `args[0]` is expr's attribute, the left subtree of + in the AST; `args[1]` is +'s attribute, the token +; `args[2]` is term's attribute, the right subtree of + in the AST.

The routine to use this subclass is straightforward:

```
def parse(tokens):
    parser = ExprParser()
    return parser.parse(tokens)
```

Although omitted for brevity, `ExprParser` can be subclassed to add grammar rules and actions, the same way that `SimpleScanner` was subclassed.

GenericParser allows a user to specify ambiguous grammars. For practical purposes, this means that GenericParser must be able to choose between multiple derivations of an input. Currently, SPARK relies upon the user to make this choice at parse time by invoking an ambiguity-resolution method when necessary. More elaborate mechanisms are known, such as disambiguating rules [1] and filters [19].

Writing actions to build ASTs for large languages can be tedious. An alternative is to use the GenericASTBuilder class instead of GenericParser, which automatically constructs the tree:

```
class AnotherExprParser(GenericASTBuilder):
    def __init__(self, AST, start='expr'):
        GenericASTBuilder.__init__(self, AST,
                                   start)

    def p_expr_1(self, args):
        ' expr ::= expr + term '
    def p_expr_2(self, args):
        ' expr ::= term '

    def p_term_1(self, args):
        ' term ::= term * factor '
    def p_term_2(self, args):
        ' term ::= factor '

    def p_factor_1(self, args):
        ' factor ::= number '
    def p_factor_2(self, args):
        ' factor ::= float '
```

(There is a more abbreviated way to express this code which we omit.) The constructor is passed the AST class, so GenericASTBuilder knows how to instantiate AST nodes.

By default, GenericASTBuilder constructs a *concrete* syntax tree which faithfully reflects the structure of the grammar. Depending on the node type being built, one of two methods inside GenericASTBuilder is invoked to construct the node: `terminal` or `nonterminal`. The user may override these with methods which shape a more compact AST rather than a concrete syntax tree.

After syntax analysis is complete, the parser has produced an AST, and verified that the input program adheres to the grammar rules. Next, the input's meaning must be checked by the semantic analyzer.

2.3. Semantic Analysis

Semantic analysis is performed by traversing the AST. Rather than spread code to traverse an AST all over the compiler, we have a single base class, GenericASTTraversal, which knows how to walk the tree. Subclasses of GenericASTTraversal supply methods which get called depending on what type of node is encountered.

To determine which method to invoke, GenericASTTraversal will first look for a method with the same name as the node type (augmented by the prefix "`n_`"), then will fall back on an optional default method if no more specific method is found. This design is a combination of the Reflection [7] and Default Visitor [25] design patterns.

Of course, GenericASTTraversal can supply many different traversal algorithms. We have found three useful: preorder, postorder, and a pre/postorder combination. (The latter allows methods to be called both on entry to, and exit from, a node.)

For example, say that we want to forbid the mixing of floating-point and integer numbers in our expressions, raising an exception if such mixing occurs:

```
class TypeCheck(GenericASTTraversal):
    def __init__(self, ast):
        GenericASTTraversal.__init__(self, ast)
        self.postorder()

    def n_number(self, node):
        node.exprType = 'number'
    def n_float(self, node):
        node.exprType = 'float'

    def default(self, node):
```

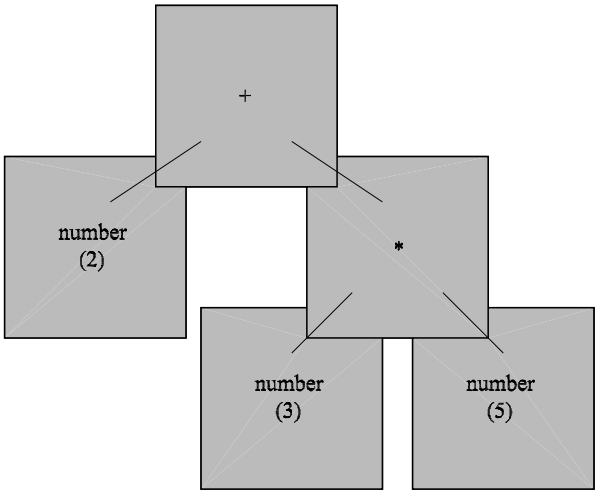

Fig. 2. Pattern covering of AST for `2 + 3 * 5`.

```
        # this handles + and * nodes
        leftType = node.left.exprType
        rightType = node.right.exprType
        if leftType != rightType:
            raise 'Type error.'
        node.exprType = leftType
```

After this phase, we have an annotated AST for an input program that is lexically, syntactically, and semantically correct — but that does nothing. The final phase, evaluation, remedies this.

2.4. Evaluation

As already mentioned, the evaluation phase can traverse the AST and implement the input program, either directly through interpretation, or indirectly by generating code for a given target language.

Our expressions, for instance, can be easily interpreted using GenericASTTraversal. For the sake of variety, we use SPARK's GenericASTMatcher class below. With GenericASTMatcher, patterns to look for in the AST are specified in a linearized tree notation, which looks remarkably like grammar rules. GenericASTMatcher determines a way to cover the AST with these patterns, then executes actions associated with the chosen patterns.

```
class Interpreter(GenericASTMatcher):
   def __init__(self, ast):
      GenericASTMatcher.__init__(self, 'V', ast)
      self.match()
      print ast.value

   def p_number(self, node):
      ' V ::= number '
      node.value = int(node.attr)
   def p_float(self, node):
      ' V ::= float '
      node.value = float(node.attr)

   def p_add(self, node):
      ' V ::= + ( V V ) '
      node.value = node.left.value
                   + node.right.value
   def p_multiply(self, node):
      ' V ::= * ( V V ) '
      node.value = node.left.value
                   * node.right.value
```

The AST covering is shown in Figure 2. In the above code, `int` and `float` are built-in functions which convert strings to integers and floating-point numbers, respectively.

The patterns specified may be arbitrarily complex, so long as all the nodes specified in the pattern are adjacent in the AST. To match both `+` and `*` nodes, for instance, this method could be added:

```
    def p_addmul(self, node):
        ' V ::= + ( V * ( V V ) ) '
        node.value = node.left.value + \
                     node.right.left.value * \
                     node.right.right.value
```

3. Experience with the Design and Implementation of SPARK

Much of the design and implementation of SPARK derives from the need to satisfy a specific target audience of users. We begin by explaining what this audience is, then discuss the repercussions of this choice in more detail.

3.1. Remember Your Audience

The advice "remember your audience" is dispensed unabashedly to speakers, writers, and web page designers. It is equally applicable to software designers. In the case of SPARK, we made two base assumptions about SPARK's users.

First, we assume that our users are programmers, who do not necessarily possess a background in compiler construction. Unlike most compiler toolkits, we deliberately hide details that are traditionally foisted onto users. The best example of this is the fact that SPARK's parser has no restrictions on the input grammar; a user need not understand parsing theory in order to use it. We have also adopted the compilation model discussed in Section 2 to target this class of users, because this provides a single template with which users can structure their compilers. (Expert users may of course abuse SPARK's classes in whatever manner they see fit.)

Second, SPARK users are thought to be occasional users. Very few people — ourselves included — design and implement domain-specific languages on a daily basis. SPARK is thus designed not just for ease of use, but for ease of occasional use. In particular, we have attempted to reduce the cognitive load associated with using SPARK by providing minimal interfaces to SPARK's classes, requiring very little of user-supplied token and AST classes, and in some cases by deliberately omitting features that could be otherwise implemented by users. Other restrictions have been avoided for the same reasons: there is no "compiler build" phase for users to worry about, making each of SPARK's classes software components which may be inserted freely into an implementation.

SPARK could be seen as a "lightweight" tool, compared to other compiler construction toolkits such as Eli [13], Gentle [29], and Cocktail [14]. SPARK supports a simple compilation model, is highly coupled to its host language, and does not require installation and use of a large suite of tools.

Obviously, some of these design guidelines benefit both groups of users, such as SPARK's close relationship with its host language, Python.

3.2. Co-evolution and Integration

SPARK has been integrated very tightly into Python. This reduces the (re)learning curve for users already familiar with Python, which is consistent with our design goals. At the same time, it makes SPARK's design vulnerable to the ever-increasing pace of Python's evolution, a problem which will eventually impact user code. Python-specific integration also limits the portability of SPARK to languages which do not possess the same features. In particular, the ability to easily associate string attributes with methods such that they may be reflectively examined is important to both the implementation and "feel" of SPARK. We are looking at C# as another possible target because of its attribute support [24].

Unfortunately, there were few other alternatives to tight integration. Having SPARK specifications in a separate file, or embedded in Python programs surrounded by special delimiters, would have violated our design constraints by necessitating a compiler-build or preprocessing phase, and decreasing the ease of occasional use. However, it would have permitted SPARK to be better isolated from changes in the host language.

There are other tradeoffs too. We look at these in terms of two key areas where SPARK is tightly coupled with Python: documentation strings and regular expressions.

3.2.1. Documentation Strings

Python allows documentation strings, or "docstrings," to be attached to methods in a straightforward fashion: a string as the first line of a method. This can be thought of as associating code for the method with meta-information about the method, a comment intended for a human reader. SPARK also uses docstrings for meta-information, but this information is intended for SPARK's use: regular expressions that describe tokens, grammar rules, and tree patterns. This choice was made because, typically, Python users are already familiar with docstrings, but also for pragmatic reasons. Docstrings, unlike other types of comments in Python code, are retained and are accessible at run-time. This aspect of docstrings is critical as

SPARK's classes rely upon it to extract needed information at run-time and to associate this information with a particular method.

When SPARK was first conceived, there were few (if any) Python packages using docstrings in this manner. Since that time, a number have arisen, resulting in the potential for semantic conflicts between packages; it is even rumored that some users write documentation in docstrings.

Recent releases of Python have included support for function attributes, which are meant to alleviate these collisions by allowing arbitrary attributes to be attached to methods. From SPARK's perspective, there are several drawbacks: existing user code would need to be changed; assignments to function attributes are statements that must be executed, unlike the declarative nature of docstrings; a new source of typographically-induced bugs is introduced (because attribute names can be mistyped); function attributes are considerably less integrated and elegant.

Whether function attributes will become the dominant way of handling meta-information in Python remains to be seen. Here, the fate of SPARK is out of our hands as a result of the tight integration with Python.

3.2.2. Regular Expresssions

In contrast, we had hoped for certain fates to be out of our hands in the area of regular expressions. Rationalizing that Python users would understand regular expressions, being an important feature of Python,[3] we decided that it would be best to have users specify tokens using Python regular expression notation as opposed to some other formalism. Implementation of SPARK's scanner then becomes trivial, a matter of concatenating the user's regular expressions together as alternatives of a single, large regular expression; throughout this section, we call this the "single" regular expression, and the ones specified by the user "component" regular expressions. As a side effect of this implementation, we automatically benefit from any speedups and other enhancements, such as Unicode support, made to Python's regular expression code.

There were some concessions to be made. Python's regular expression engine, and hence SPARK's scanner class, is restricted to using strings as input; the entire input file must be read into a string prior to being tokenized. This is not a terrible burden for many domain-specific language applications, but severely limits SPARK's usefulness for, say, processing large XML files.

Another issue lay with the semantics of Python regular expressions. Python follows other scripting languages like Perl and Tcl in supplying "first-then-longest" semantics for regular expressions.[4] This means that the first match for a regular expression will be taken, not necessarily the longest match, contrary to what users might expect from a scanner. As SPARK users do not even see the single regular expression that SPARK constructs on their behalf, first-then-longest semantics are especially troubling because the user will not know which of their component regular expressions appeared first. At first glance, one might think that SPARK could use the static position of the user's regular expressions in their source code to order the single regular expression. One would be wrong. Internally, Python compiles methods, and stores method names into a hashed data structure; all information about method ordering is lost. SPARK offers three workarounds to this problem: a user's component regular expression is never broken apart, so a user may combine and order conflicting alternatives in one component regular expression; component regular expressions in a subclass are placed before those of a superclass in SPARK's single regular expression; users may define a default method whose component regular expression is placed after all those in the default method's class.

We have found that SPARK has not gained as much from speedups to Python's regular expression engine as anticipated. The first-then-longest semantics are implemented by a linear search through the alternatives. As SPARK's single regular expression is atypical in that it tends to be very long with many alternatives,

[3] Regular expressions are prominent in Python code, but they are not as tightly integrated into the language as they are in Perl.
[4] Versions 8.1 and later of Tcl have different semantics.

we suffer a corresponding performance penalty. In one case, a handwritten scanner interpreted by the Python interpreter was 3.1 times faster than the corresponding SPARK scanner, even though Python's regular expression engine is implemented in C!

3.3. The Price of Generality and Code Reuse

Other performance problems have haunted SPARK, also due to design decisions. As mentioned, there are no restrictions on the input grammar to SPARK's parser. Any context-free grammar, even an ambiguous one, may be supplied by the user. To parse with these grammars, we use Earley's general parsing algorithm [10, 11].

Using general parsing algorithms is a somewhat unusual choice in that they tend to have more overhead than more specialized algorithms like the LALR(1) algorithm used in Yacc [18, 22]. Such a choice is not completely unprecedented, however; generalized LR parsing, another general parsing algorithm, is used in the ASF+SDF toolkit [31]. SPARK incorporates Earley's algorithm rather than generalized LR parsing, primarily because no precomputation is required for Earley's algorithm.[5] Earley's algorithm suffers from two major problems, from SPARK's perspective: the performance problem and the delayed action problem.

Addressing the performance problem is a matter of ongoing research. We have made Earley's algorithm run in time comparable to that of much more specialized algorithms, by generating code to parse a specific grammar and creating a directly-executable Earley parser [4]. This is not suitable for inclusion in SPARK, unfortunately, due to the large amounts of precomputation required. More promisingly, we have devised a new type of automaton tailored for fast Earley parsing, the states of which can be unobtrusively constructed as needed [3, 5]. We are currently incorporating this automaton into SPARK. Preliminary results show that use of our new automaton in SPARK can result in a 40% speed increase.

The delayed action problem refers to the fact that general parsing algorithms, in general, must read and verify their entire input first [16]. As a consequence of this, the user's semantic actions associated with grammar rules may not be executed until after the input is recognized. Even simple semantic actions, such as inserting names into symbol tables, must be deferred. While we have looked at ways to remedy this problem [3], SPARK tries to minimize the impact of the delayed action problem through facilities for automatically constructing parse trees, something semantic actions are commonly used for.

Besides automatically constructing parse trees, the generality of Earley's algorithm lends itself to use for tree-pattern matching, by which SPARK users can implement the evaluation phase of a compiler. This is known as Graham/Glanville code generation [12], and use of Earley's parsing algorithm in a Graham/Glanville code generator is known to correct problems with the technique [8]. The interpreter from Section 2.4, for example, would correspond to the grammar:

```
V ::= ) V V ( +
V ::= ) V V ( *
V ::= ) ) V V ( * V ( +
V ::= number
V ::= float
```

The AST in Figure 2 would be linearized to `) ) number number ( * number ( +`, with parentheses inserted to delineate the AST's structure. Conflicting matches show up as ambiguities during parsing, and are handled using GenericParser's ambiguity resolution mechanism.

There is a great dependency in SPARK on the implementation of Earley's algorithm in GenericParser, as it is re-used for various purposes. Figure 3 shows the class structure of the user-visible classes of SPARK, along with their key methods. Our experience is that this serves to amplify the performance problem, and this is why we have devoted much effort towards solving that problem.

[5] Although implementations of generalized LR parsing which compute information on the fly are possible [17], they are more complicated than the implementation of Earley's algorithm.

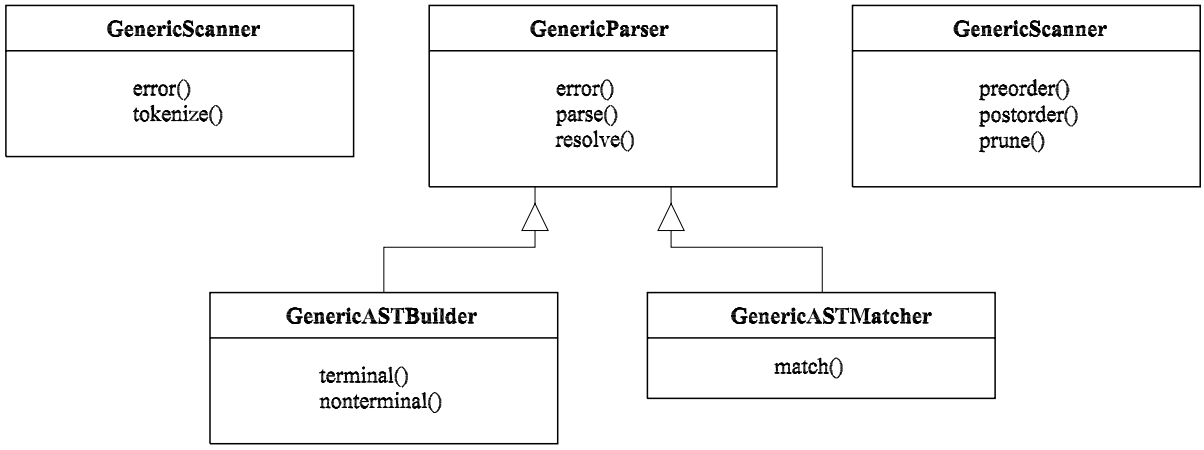

Fig. 3. SPARK's class structure.

3.4. Serendipity, Schrödinger, and Speed

Our criterion that user-supplied tokens should have a minimal interface led us to an accidental discovery. SPARK's parser assumes that tokens are objects and, while parsing, compares them to the names of terminal symbols using Python's == operator. This in turn causes the token's user-supplied `__cmp__` method to be invoked to perform the comparison. In this way SPARK never needs to know how or where the token's type is stored — the user need only to supply the one "comparison method" in their token class.

An odd question arose: what if the user's comparison method lied? When asked by the parser, what if a token claimed to not only have its token type, but other token types as well? The result is what we call a Schrödinger token, which has some very useful applications, as well as providing a means of avoiding the delayed action problem. A more complete discussion[6] appears in [6]; we summarize the ideas here.

For simplicity, consider processing a file which contains key/value pairs. The problem is that the type of tokens may be context-sensitive, because the name of a key may appear as a value:

```
foo  123
bar  456
baz  foo
```

Ideally, we would like users to be able to write a grammar which accurately reflects the structure of this small domain-specific language, as shown in Figure 4. In this grammar, `foo`, `bar`, `baz`, and `value` are all distinct token types, and therein lies the problem. Recall that SPARK's scanner is finished operation by the time the parser starts; the scanner does not have access to context information from the parser to determine when `foo` is a key and when it is a `value`.

```
pairs ::= pairs pair
pairs ::= pair
pair  ::= foo value
pair  ::= bar value
pair  ::= baz value
```

Fig. 4. An ideal grammar for key/value pairs.

By making a one-line change to their `__cmp__` method, the problem is easily solved. The token — `foo` in this case — can claim to be *both* token types, effectively giving the token a superposition of types. Earley's parsing algorithm essentially simulates nondeterminism, and can handle a token that appears to have more than one type; the conflict is automatically resolved once enough context information has been seen. The user does not need to make any modifications to the grammar whatsoever, unlike other techniques to solve the problem of context-dependent tokens.

Analogy can be established between these tokens with a superposition of types and Schrödinger's Cat, which was in a superposition of "alive" and "dead" states [28]. The general parser acts as the observer which resolves the actual type of the "Schrödinger token."

[6] Including a full discussion of other approaches, such as scannerless parsing [27, 32].

Unfortunately, the performance problem rears its ugly head again. We can make SPARK's parser run 30% faster by knowing an exact type for tokens, rather than having to query each token's `__cmp__` method repeatedly. In recent releases of SPARK we have done this, but the utility of Schrödinger's tokens is so great that we have taken pains to ensure that they are still supported in SPARK.

3.5. Run-time Limitations

While we gained a great deal of portability across platforms and ease of implementation working in Python, there were also some things we lost by working in an interpreted scripting language. All of these tended to play out at SPARK's run-time. It was, for example, extremely difficult to gauge the effect of a potential optimization to SPARK until such time as it was actually implemented. A number of times, the overhead required for Python to execute the "optimized" code negated any speedup that would have been obtained.

The design decision to eschew a compiler build phase and make SPARK reflectively gather information from user classes at run-time has been effective from the user standpoint, but has also severely limited what we can do in terms of precomputation. Computation done by SPARK can only be done at run-time, when speed is already critical; we must search for effective yet computationally lightweight improvements.

4. Future Directions

Some future work on SPARK is mundane but necessary, such as writing more user documentation. It would also be interesting to evaluate ease-of-use from an experimental, rather than an anecdotal, perspective. Other future directions, once important, have been abandoned. Initially, we envisioned that SPARK's parsing engine could be replaced with another, using a more efficient parsing algorithm [2]; however, the general parsing algorithm has proven itself so simple to use that its replacement is no longer being considered.

Improvements to the pattern matcher are likely. Currently, "sparse" patterns cannot be specified without giving patterns to match the entire AST. Removing this limitation would permit use of pattern matching for selective tree rewriting and possibly for some tree-based optimizations.

The scanner should also see replacement. Relying on Python's regular expression engine has unfortunately proven to be a disaster from the performance point of view. Finally, a facility for performing static verification of SPARK specifications would be useful to trap some common user errors.

One obvious direction, which we are *not* considering at this time, is the use of attribute grammars [20]. While this would alleviate the need for the user to order AST traversals for semantic checking, it would also impose an additional barrier to use by SPARK's intended audience. We hold the somewhat controversial view that the key to wider acceptance is less formalism, not more.

5. Conclusion

SPARK has been used to implement a number of domain-specific languages in Python. Targeting the toolkit to occasional users and programmers without a compiler construction background has had profound effects in both the design and implementation of SPARK.

Like most designs, a series of tradeoffs is involved. Tight integration with Python lowers the learning curve, but leaves SPARK at the mercy of ongoing Python development; re-use of regular expression and parsing engines impacts performance; avoiding a compiler build phase pushes all computation work to run-time. Nevertheless, we are pleased with the result and plan to bring SPARK's ideas to a wider audience.

Acknowledgments

Shannon Jaeger and the anonymous referees made many helpful comments on this paper. This work has been supported in part by a grant from the National Sciences and Engineering Research Council of Canada.

References

[1] A. V. AHO, S. C. JOHNSON, AND J. D. ULLMAN, Deterministic parsing of ambiguous grammars. *Communications of the ACM*, 18(2):441–452, August 1975.

[2] J. AYCOCK, Compiling little languages in Python. In *Proceedings of the 7th International Python Conference*, pages 69–77, 1998.

[3] J. AYCOCK, *Practical Earley Parsing and the SPARK Toolkit*. PhD thesis, University of Victoria, 2001.

[4] J. AYCOCK AND N. HORSPOOL, Directly-executable Earley parsing. In *Proceedings of the 10th International Conference on Compiler Construction*, pages 229–243, 2001.

[5] J. AYCOCK AND R. N. HORSPOOL, Practical Earley parsing. Submitted for publication.

[6] J. AYCOCK AND R. N. HORSPOOL, Schrödinger's token. *Software — Practice and Experience*, 31(8):803–814, 2001.

[7] F. BUSCHMANN, R. MEUNIER, H. ROHNERT, P. SOMMERLAD, AND M. STAL, *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.

[8] T. W. CHRISTOPHER, P. J. HATCHER, AND R. C. KUKUK, Using dynamic programming to generate optimized code in a Graham-Glanville style code generator. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 25–36, 1984.

[9] P. F. DUBOIS, *Pyfort Reference Manual*. Lawrence Livermore National Laboratory, sixth edition, 2000.

[10] J. EARLEY, *An Efficient Context-Free Parsing Algorithm*. PhD thesis, Carnegie-Mellon University, August 1968.

[11] J. EARLEY, An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

[12] R. S. GLANVILLE AND S. L. GRAHAM, A new method for compiler code generation. In *5th Annual ACM Symposium on Principles of Programming Languages*, pages 231–240, 1978.

[13] R. W. GRAY, V. P. HEURING, S. P. LEVI, A. M. SLOANE, AND W. M. WAITE, Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, February 1992.

[14] J. GROSCH AND H. EMMELMANN, A tool box for compiler construction. Technical Report 20, GMD, January 1990.

[15] D. GRUNE, H. E. BAL, C. J. H. JACOBS, AND K. G. LANGENDOEN, *Modern Compiler Design*. Wiley, 2000.

[16] D. GRUNE AND C. J. H. JACOBS, *Parsing Techniques: A Practical Guide*. Ellis Horwood, 1990.

[17] J. HEERING, P. KLINT, AND J. REKERS, Incremental generation of parsers. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 179–191, 1989.

[18] S. C. JOHNSON, YACC — yet another compiler. *UNIX Programmer's Manual, 7th Edition*, 2B, 1978.

[19] P. KLINT AND E. VISSER, Using filters for the disambiguation of context-free grammars. In *Proceedings of the ASMICS Workshop on Parsing Theory*, pages 1–20, 1994.

[20] D. E. KNUTH, Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Errata in [21].

[21] D. E. KNUTH, Semantics of context-free languages: Correction. *Mathematical Systems Theory*, 5(1):95–96, 1971.

[22] J. R. LEVINE, T. MASON, AND D. BROWN, *Lex & Yacc*. O'Reilly & Associates, second edition, 1992.

[23] M. R. LEVY, Web programming in Guide. *Software — Practice and Experience*, 28(11):1581–1603, 1998.

[24] Microsoft Corporation, *C# Language Specification, version 0.28*, 2001.

[25] M. E. NORDBERG III, Variations on the visitor pattern. In *Collected Papers from the PLoP '96 and EuroPLoP '96 Conferences*. Washington University, 1997. Technical Report WUCS-97-07.

[26] E. S. RAYMOND, The CML2 language: Python implementation of a constraint-based interactive configurator. In *Proceedings of the 9th International Python Conference*, pages 135–142, 2001.

[27] D. J. SALOMON AND G. V. CORMACK, Scannerless NSLR(1) parsing of programming languages. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 170–178, 1989.

[28] E. SCHRÖDINGER, Die gegenwärtige situation in der quantenmechanik. *Die Naturwissenschaften*, 23:807–812,823–828,844–849, 1935. Translation in Trimmer [30].

[29] F. W. SCHRÖER, *The GENTLE Compiler Construction System*. R. Oldenbourg Verlag, 1997.

[30] J. D. TRIMMER, The present situation in quantum mechanics: a translation of Schrödinger's "cat paradox" paper. *Proceedings of the American Philosophical Society*, 124(5):323–338, 1980.

[31] A. VAN DEURSEN, Introducing ASF+SDF using the λ-calculus as example. In *Executable Language Definitions*. PhD thesis, University of Amsterdam, 1994.

[32] E. VISSER, Scannerless generalized-LR parsing. Technical Report P9707, University of Amsterdam Programming Research Group, 1997.

[33] R. L. WHITE AND P. GREENFIELD, Using Python to modernize astronomical software. In *Proceedings of the 8th International Python Conference*, pages 103–109, 2000.

Received: July, 2001
Revised: October, 2001
Accepted: November, 2001

Contact address:
John Aycock
Department of Computer Science
University of Calgary
2500 University Drive N.W.
Calgary, Alberta, Canada T2N 1N4
e-mail: aycock@cpsc.ucalgary.ca

JOHN AYCOCK received a B.Sc. (1993) from the University of Calgary, and a M.Sc. (1998) and Ph.D. (2001) from the University of Victoria, all in computer science. He has recently returned to Calgary to become Assistant Professor at the University of Calgary. His research interests are in compilers, compiler tools, system software, and operating systems.