An Overview of SORCERER:

A Simple Tree Parser and Translator Generator

Terence Parr
Parr Research Corporation
(in cooperation with Univ. of Minnesota)
Minneapolis MN

Aaron Sawdey
Univ. of Minnesota
Army High Performance Comp. Res. Center

Gary Funck
Intrepid Technology, Inc.
Mountain View CA

What Are We Trying To Solve?

1. Given a tree data structure, generate output or execute a specific sequence of instructions.

2. Given a tree, generate a transformed version of that tree.

Where would one want to do that?

Source-to-source translators, compilers, interpreters, data format converters.

Common Translation Strategy

A parser (hand-built or using `lex/yacc`) constructs trees that are traversed by one of:

1. Hand-built, hard-coded tree walker.
2. Hand-built tree walker using general library routines such as "iterators."

Disadvantages:

Resulting translators are difficult to maintain and modify; source code is tedious to write and hard to read. `yacc` has no automatic tree construction facility.

Automated Solutions?

There are innumerable translation systems; e.g., Eli, Cornell Program Synthesizer, TXL, COCOL, Ox, Puma, DELTA, TWIG, BEG.

They are not very satisfying because they:

- Are slow.
- Lock the programmer into specific environment (integration also tough).
- Require a specific front end.
- Are language-specific.
- Require the use of tool-specific language.
- Restrict placement of actions.

What is SORCERER Good At?

1. Translators for "little Languages;" descriptions are flexible, extensible, and reasonably terse.

2. Large, complex translators that must run quickly.

SORCERER is **not** good at "one-liners" like `sed`, `awk`, or `perl`. However, tools that are really good at "one-liners" tend to be terrible at "1000 liners."

A Few Current Applications

- Fortran-P; research project at AHPCRC.
- Cybil to C; commercial project.
- Pascal to Ada; commercial project.
- C+-; example for ANTLR/SORCERER book.
- ANVIL graphics language; commercial project.
- Theorem prover; poor man's Prolog.

Features

- Simple, flexible, “lightweight.”

- Generates fast, top-down, tree parsers in C/C++.

- Easily integrated into other applications–does not rely on any particular intermediate form, parser generator, or other pre-existing application.

- Supports decent tree-rewriting; growing library of tree manipulation routines.

- Public domain software.

Introductory Example:

Child-sibling notation	Graphical form
(`ASSIGN` lhs rhs)	$\begin{tikzcd} \mathtt{ASSIGN} \arrow[d] & \\ lhs \arrow[r] & rhs \end{tikzcd}$
`( ASSIGN ID (PLUS FLOAT ID))`	$\begin{tikzcd} \mathtt{ASSIGN} \arrow[d] & & \\ \mathtt{ID} \arrow[r] & \mathtt{PLUS} \arrow[d] & \\ & \mathtt{FLOAT} \arrow[r] & \mathtt{ID} \end{tikzcd}$

IR Form for C Assignments

```
assign(AST *t)
{
    if ( t->token==ASSIGN ) {
        expr(t->down);
        printf(" := ");
        expr(t->down->right);
    }
    else error;
}
```

C Code to Translate C Assignments

```
assign  :   #( ASSIGN expr <<printf(" := ");>> expr )
        ;
```

Equivalent SORCERER Description

Programming Interface

- Define `SORAST`.

- Define member functions `down()` and `right()` for C++; C requires actual `down` and `right` fields.

- Define member function `token()` (field `token` in C); used to distinguish between trees with same structure, but different contents:

$$\begin{tikzcd} A \arrow[d] & \text{and} & D \arrow[d] \\ B & & E \end{tikzcd}$$

Can handle any tree structure given these constraints.

Notation:

Item	Description/Example
leaf	token type `ID`
`.`	wild card `#( FUNC ID (.)* )`
rule-name	reference to another rule: `expr`
`#(...)`	tree pattern `#(IF expr slist slist)`
`<<...>>`	user-defined semantic action `<<printf("%s", t->name);>>`
`(...)`	subrule `("int" | ID | storage_class)`
`(...)*`	closure: `ID ("," ID)*`
`(...)+`	positive closure `slist :  ( stat | SEMICOLON )+ ;`
`{...}`	optional: `{ ELSE stat }`

`rule :` $alternative_1$
`     |` $alternative_2$
`       ...`
`     |` $alternative_n$
`     ;`

Simple Translation Example:

Infix to Postfix

```
#header <<
#include "my_ast_def.h"
>>

<<
main()
{
    STreeParser myparser;
    STreeParserInit(&myparser);
    e(&myparser, &some_input_tree);
}
>>

e   :   #( Plus e e )  <<printf("\tadd\n");>>
    |   #( Mult e e )  <<printf("\tmult\n");>>
    |    t:INT          <<printf("%d\n",  t->ival);>>
    ;
```

C++ Interface

```
#header <<
class SORAST : public myAST, public SORASTBase {
     define SORCERER interface: right(), etc...
};
>>

<<
main()
{
    InfixToPostfixTranslator myparser;
    myparser.e(&some_input_tree);
}
>>

class InfixToPostfixTranslator {
e   :   #( Plus e e )  <<printf("\tadd\n");>>
    |   #( Mult e e )  <<printf("\tmult\n");>>
    |   t:INT          <<printf("%d\n", t->ival);>>
    ;
}
```

Tree Rewriting

With `-transform` command-line option:

1. Every rule has an input and an output output tree.

2. The default rewrite action for each rule is to copy input to output.

3. In actions, *label* refers to an input node and `#`*label* refers to an output node. `#`*rule* refers to result tree for `rule`.

4. `#[...]` and `#(...)` are node and tree constructors.

5. The “`!`” symbol is used to override the default action.

Rewriting Example:

Swap operand order

```
#header <<
#include "my_ast_def.h"
>>

<<
main()
{
    SORAST *result;
    STreeParser myparser;
    STreeParserInit(&myparser);
    expr(&myparser, &some_input_tree, &result);
}
>>

expr!:  #(a:Plus b:expr c:expr) <<#expr = #(#a,#c,#b);>>
    |   #(a:Mult b:expr c:expr) <<#expr = #(#a,#c,#b);>>
    |   v:Int                 <<if ( f(v)) #expr = #v;>>
    ;
```

Note the use of #*label* versus *label*.

Yet More Examples:

Mixed default and explicit rewriting

```
expr:   #(Plus expr expr)
    |!  #(a:Mult b:expr c:expr) <<#expr = #(#a,#c,#b);>>
    |   Int
    ;
```

Filtering out second operand of multiplies

```
expr:   #(Plus expr expr)
    |   #(Mult expr expr!)
    |   Int
    ;
```

Sample Library Routines:

Is tree an assignment?

```
AST *lhs, *rhs;
int n;
n = ast_scan("#( ASSIGN %1:. %2:.)",mytree,&lhs,&rhs);
if ( n!=2 ) printf("not assignment");
```

Perform an operation on every scalar assignment in `slist`.

```
AST *cursor = slist, *p,
    *template = #( #[ASSIGN], #[ID] );
while ((p=ast_find_all(slist, template, &cursor)))
{
     some_operation(p);
}
```

Conclusion

1. Simple tree-parser and translator generator developed by programmers for programmers.

2. SORCERER is powerful, flexible, easy to understand and easy to integrate into other applications.

3. Can operate with virtually any tree structure. More flexible than common strategy and more practical than other translator tools.

Public domain: `everest.ee.umn.edu` in `pub/pccts/sorcerer`.

Supported and being upgraded; WWW: `http://tempest.ecn.purdue.edu:8001/`.

Newsgroup: `comp.compilers.tools.pccts`.

What the Output Actually Looks Like

```
void assign(STreeParser *_parser, AST **_root)
{
    AST *_t = *_root;
    if ( _t!=NULL && (_t->token==ASSIGN) ) {
        {_SAVE;
        _MATCH(ASSIGN); _DOWN;
        expr(_parser, &_t);
        if ( !_parser->guessing ) {
            printf(":=");
        }
        expr(_parser, &_t);
        _RESTORE;}
        _RIGHT;
    }
    else {
        if ( _parser->guessing ) _GUESS_FAIL;
        no_viable_alt(_parser, "assign", _t);
    }
    *_root = _t;
}
```

Tree Description	Tree Structure
#(A B C D E)	$$\begin{tikzcd} A \arrow[d] & & & \\ B \arrow[r] & C \arrow[r] & D \arrow[r] & E \end{tikzcd}$$
#(A (B C D) E)	$$\begin{tikzcd} A \arrow[d] & & \\ B \arrow[d] \arrow[rr] & & E \\ C \arrow[r] & D & \end{tikzcd}$$
#(A (B C) (D E))	$$\begin{tikzcd} A \arrow[d] & \\ B \arrow[d] \arrow[r] & D \arrow[d] \\ C & E \end{tikzcd}$$

Sample Tree Specification and
Graphical Representation Pairs

`#(If expr slist { slist })`: optional subrule.

Possible Tree Structures

$$\begin{tikzcd}\mathtt{If} \arrow[d] & & \\ expr \arrow[r] & slist \arrow[r] & slist\end{tikzcd}$$

$$\begin{tikzcd}\mathtt{If} \arrow[d] & \\ expr \arrow[r] & slist\end{tikzcd}$$

`#(Call Subroutine (expr)*)`: zero-or-more-times subrule.

Possible Tree Structures

$$\begin{tikzcd}\mathtt{Call} \arrow[d] \\ \mathtt{Subroutine}\end{tikzcd}$$

$$\begin{tikzcd}\mathtt{Call} \arrow[d] & \\ \mathtt{Subroutine} \arrow[r] & expr\end{tikzcd}$$

$$\begin{tikzcd}\mathtt{Call} \arrow[d] & & \\ \mathtt{Subroutine} \arrow[r] & expr \arrow[r] & expr\end{tikzcd}$$

Semantic Predicates

How to disable certain expression elements when matching the left hand side of an assignment.

```
<<enum SIDE { LHS, RHS };>>

stat:   #( ASSIGN expr[LHS] expr[RHS] )

    |   ...

    ;

expr[int side]

    :   <<side!=LHS>>? INT     <<action_1>>

    |   ID                     <<action_2>>

    ;
```

Syntactic Predicates

How to distinguish between two tree patterns with common, left-prefixes of arbitrary length.

```
expr:   ( #(Minus . .) )?  #( Minus expr expr )
    |                      #( Minus expr )
    ;
```

Semantic and syntactic predicates behave just as in ANTLR.

Another Example: eval expression

```
#header <<
#include "my_ast_def.h"
>>

<<
main()
{
    int result;
    STreeParser myparser;
    STreeParserInit(&myparser);
    result = eval(&myparser, &my_input_tree);
    printf("result is %d\n", result);
}
>>

eval > [int r]
    :   <<int opnd1, opnd2;>>
        #(Plus eval>[opnd1] eval>[opnd2])
        <<r = opnd1+opnd2;>>

    |   <<int opnd1, opnd2;>>
        #(Mult eval>[opnd1] eval>[opnd2])
        <<r = opnd1*opnd2;>>

    |   v:Int <<r = v->val;>>
    ;
```

Error Detection

- **`mismatched_token()`**: couldn't find a desired token.
- **`mismatched_range()`**: couldn't find token in desired range.
- **`missing_wildcard()`**: found an unexpected `NULL` pointer.
- **`no_viable_alt()`**: none of the alternative productions matched current tree.