An Overview of SORCERER:

A Simple Tree-Parser Generator

Terence Parr and *Aaron Sawdey*
Univ. of Minnesota
Army High Performance
Comp. Res. Center

Gary Funck
Intrepid Technology, Inc.
Mountain View CA

Why SORCERER?

- Optimal tree walks of code-generator generators and the powerful attribute evaluation schemes of source-to-source translator systems are overkill.

- Programmers would rather avoid the overhead and complexity because they simply want to traverse their trees and execute a few actions.

- SORCERER is suitable for class of problems lying between code-generator generators and full source-to-source translator generators.

Features:

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

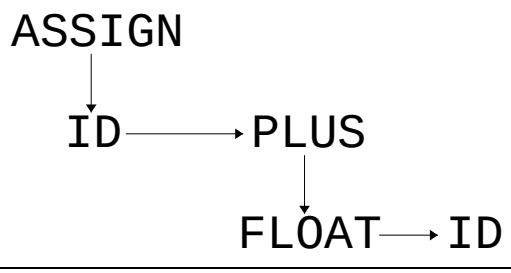
- Accepts extended BNF notation

- Allows predicates to direct the tree walk with semantic and syntactic context information.

- Does not rely on any particular intermediate form, parser generator, or other pre-existing application.

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

C→Pascal Example:

IR Child-sibling form	Graphical form
`#(ASSIGN ID (PLUS FLOAT ID))`	

By hand: (from `a=3.4+b` to `a:=3.4+b`)

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

Versus:

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

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;
}
```

Programmer's Interface

1. The type of an input tree node must be `AST`.

2. The trees must be in child-sibling form with `right` and `down` pointers; C++ makes relaxes this constraint.

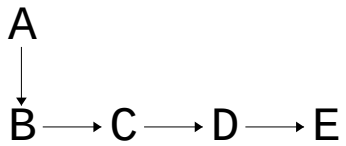
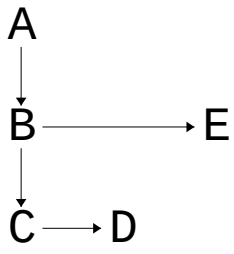
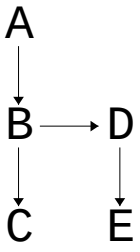
3. A `token` field must exist, which is used to recognize tree contents. For example, the `token` is used to distinguish between

   which have the same structure, but different contents.

Notation

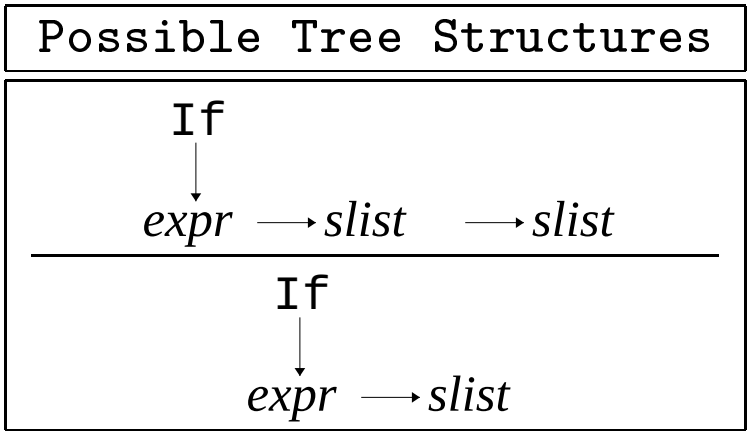
What	Example
rule	`a, varName`
token/token class	`ID, REGISTERS`
regular expression	`"do" "[a-z]+"`
wildcard	`#(OP . .)`
tree pattern	`#(IF expr slist slist)`
optional	`{else stat}`
zero-or-more	`ID ( "," ID )*`
one-or-more	`(stat)+`
action	`<< i++; >>`
semantic predicate	`<<isType(LATEXT(1))>>?`
syntactic predicate	`(declaration)?`

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

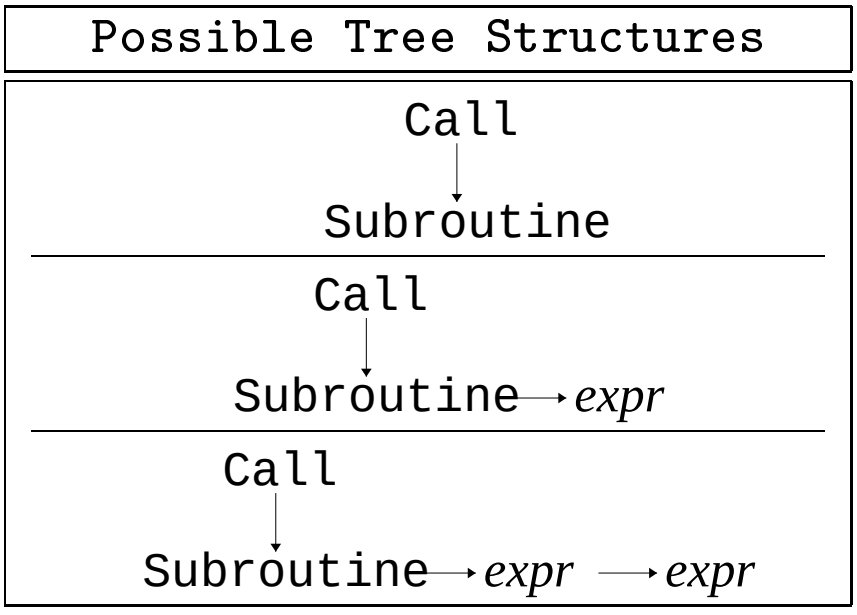
Tree Description	Tree Structure
#(A B C D E)	
#(A (B C D) E)	
#(A (B C) (D E))	

Sample Tree Specification and
Graphical Representation Pairs

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



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



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.

Simple Example: RPN

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

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

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

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;>>
    ;
```

Translation Support:

- Element labels:

```
a : #( DO u:ID expr expr #( v:SLIST (stat)* ) )
       <<printf("induction var is %s\n", u->symbol);
          analyze_code(v);>>
    ;
```

- Arguments, return values, local vars:

```
rule[args] > [ret_val]
    : <<int local;>> alt1 | alt2 | ... | altn ;
```

"Transform" mode:

- There exists an input and an output tree for every rule.

- If given no instructions, SORCERER "copies" input to output.

- Can filter input trees with simple grammar annotations.

```
stats : ( stat ";"! )* ;
```

- Can modify input tree explicity:

```
assign : #(a:ASSIGN e:expr id:ID)
         <<ast_return(#(a, id, e));>>

       ;
```

Sample Library Routines:

"Give me a list of scalars on the left side of an assignment."

```
SList *find_scalar_assigns(AST *slist)
{
    SList *scalars = NULL;
    AST *cursor = slist, *p,
        *template = #( #[ASSIGN], #[ID] );
    while ((p=ast_find_all(slist, template, &cursor)))
    {
        slist_add(&scalars, p);
    }
    return scalars;
}
```

"Is tree an assignment?":

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

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.

Current Usage

- There were 457 "pings" of ftp site within first week of release.

- AHPCRC has used SORCERER since 1992 for translation of large scientific FORTRAN programs.

- Example commercial application: Pascal→ADA translator; `gary@intrepid.com`.

- Example "research" application: Theorem prover; `helz@ecn.purdue.edu`.

7. Conclusions

1. Simple tree-parser generator—useful for real-world translations.

2. SORCERER follows PCCTS philosophy of providing powerful, flexible, tools that are easy to understand.

3. Supported and being upgraded.

4. Public domain: `marvin.ecn.purdue.edu` in `pub/pccts/sorcerer`;

5. WWW: `http://tempest.ecn.purdue.edu:8001/`.

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