Compilers tutorial IV: Abstract syntax

Yacc is a powerful tool for generating parsers, by transforming formal grammars into executable code. In this tutorial we explore how to construct an abstract syntax tree (AST), which is a kind of parse tree which discards irrelevant details while fully preserving the meaning of the original program.

A bit of theory: A syntax-directed translation (SDT) scheme consists of a grammar with attached program fragments (called *actions*). Whenever a *production* is used, during syntax analysis, its *action* is executed. One of the main uses of these schemes is to build syntax trees. Every time a production is used during bottom-up parsing, the corresponding action can *create* new nodes and/or *relate* children nodes to the parent.

Abstract syntax trees

Trees can be represented in a number of ways. The data structures below specify that a *node* in the AST has a linked list of *children nodes*:

```
struct node {
    enum category category;
    char *token;
    struct node_list *children;
};

struct node_list {
    struct node *node;
    struct node_list *next;
};
```

Every node has a syntactic *category* denoting a specific programming construct occurring in the input program, such as a function, a parameter declaration, or a natural constant:

```
enum category {Program, Function, ..., Identifier, Natural, ...};
```

Every node also includes a pointer to the original *token*. The token is necessary because a node of category `Identifier` must have the name of the function or variable, a node of category `Natural` must have the string of digits representing the natural constant, and so on.

To build an AST we only need two operations: creating a new node and adding a child node to a parent node. Two functions provide that functionality:

```
struct node *newnode(enum category category, char *token);
void addchild(struct node *parent, struct node *child);
```

The first function returns a newly allocated node with all its fields initialized (including an empty list of children). The second function appends a node to the list of children of the parent node. They are provided in files `ast.h` and `ast.c`.

Syntax-directed translation

During bottom-up parsing, an action `{...}` is executed when the corresponding production is used. The combined result of all those executions is the AST.

Consider the following *yacc* specification (which you will complete). Notice that it is included in file `petit.y`, that you should carefully analyze.

```
program: IDENTIFIER '(' parameters ')' '=' expression
                { $$ = program = newnode(Program, NULL);
                  struct node *function = newnode(Function, NULL);
                  addchild(function, newnode(Identifier, $1));
                  addchild(function, $3);
                  addchild(function, $6);
                  addchild($$, function); }
    ;
parameters: parameter              { /* ... */ }
    | parameters ',' parameter     { /* ... */ }
    ;
parameter: INTEGER IDENTIFIER      { /* ... */ }
    | DOUBLE IDENTIFIER            { /* ... */ }
    ;
arguments: expression              { /* ... */ }
    | arguments ',' expression     { /* ... */ }
    ;
expression: IDENTIFIER             { /* ... */ }
    | NATURAL                      { /* ... */ }
    | DECIMAL                      { /* ... */ }
    | IDENTIFIER '(' arguments ')' { /* ... */ }
    | IF expression THEN expression ELSE expression  %prec LOW
                                   { /* ... */ }
    | expression '+' expression    { /* ... */ }
    | expression '-' expression    { /* ... */ }
    | expression '*' expression    { /* ... */ }
    | expression '/' expression    { /* ... */ }
    | '(' expression ')'           { $$ = $2; }
    ;
```

When the first production is used, the right-hand side contains a *function* with its `parameters` and `expression`. Parsing will be finishing there. The corresponding action executes 6 statements, in the following order: *(1)* the AST's root node `Program` is created; *(2)* a new `Function` node is created; *(3)* a new `Identifier` node is created, with the function name, and becomes a child of the `Function` node; *(4)* the `parameters` node `$3` becomes a child of the `Function` node; *(5)* the `expression` node `$6` becomes a child of the `Function` node; and *(6)* the new `Function` node becomes a child of the `Program` node.

Token types and `%union`

As seen in the previous tutorial, the semantic value of a token must be stored in global variable `yylval`, which has type `int` by default. Token declarations use:

```
%token INTEGER DOUBLE IF THEN ELSE
```

When we need to use multiple data types, the `%union` declaration allows us to specify the distinct types that might be stored in `yylval`. In our case:

```
%union{
    char *token;
    struct node *node;
}
```

This `%union` declaration modifies the type of `yylval` so that it may hold a *token* (`char *`) or a *node* (`struct node *`). In other words, `yylval` might be a string or an AST node, and a C union encapsulates the two alternatives.

Then, when we declare a token, the C type is specified as follows:

```
%token<token> IDENTIFIER NATURAL DECIMAL
```

Identifiers, naturals and decimals require their semantic value (the `char *` to the original string) to be stored. The *lex* specification should copy the semantic value by executing `yylval.token = strdup(yytext);` before returning any of these tokens.

Furthermore, syntactic variables (i.e., nonterminals) are specified using the `%type` declaration:

```
%type<node> program parameters parameter arguments expression
```

This way, the type of `$$`, `$1`, `$2`, etc., is correctly handled during parsing.

Exercises

Begin by carefully examining the file `petit.y` and the AST construction functions in files `ast.h` and `ast.c`.

1. Complete the actions marked with `/* ... */` so that an AST is constructed, for each program, using the supplied functions `newnode(...)` and `addchild(...)`.

2. Write a function to recursively traverse the AST and show its content. The goal is to call that function immediately after `yyparse()` to check that the AST is correct. Consider the following pseudocode:

   ```
   show(struct node *node, int depth) {
       print(node->category, node->token, depth)
       foreach child in node->children show(child, depth+1)
   }
   ```

Taking `factorial(integer n) = if n then n * factorial(n-1) else 1` as input, the solution to exercises 1 and 2 should have the following output:

```
Program
__Function
____Identifier(factorial)
____Parameters
______Parameter
________Integer
________Identifier(n)
____If
______Identifier(n)
______Mul
________Identifier(n)
________Call
__________Identifier(factorial)
__________Arguments
____________Sub
______________Identifier(n)
______________Natural(1)
______Natural(1)
```

3. Modify the grammar to allow for multiple functions, using the productions that follow, and implement the necessary action to construct the AST.

```
program: IDENTIFIER '(' parameters ')' '=' expression
       | program IDENTIFIER '(' parameters ')' '=' expression
```

Test your solution with the following example, found in file `factorial.pt`:

```
factorial(integer n) = if n then n * factorial(n-1) else 1
main(integer i) = write(factorial(read(0)))
```

The file `factorial.ast` contains the expected AST for this program.

Author

Raul Barbosa (University of Coimbra)

References

Aho, A. V. (2006). Compilers: Principles, techniques and tools, 2nd edition. Pearson Education.

Levine, J. (2009). Flex & Bison: Text processing tools. O'Reilly Media.

Niemann, T. (2016) Lex & Yacc. https://epaperpress.com/lexandyacc

Barbosa, R. (2023). Petit programming language and compiler. https://github.com/rbbarbosa/Petit