Compiler Construction

Tools for AST

Developing a new languge

Developing a new language is hard!

BUT hopefully tools exist!

Treecc

“ *The approach that we take with ”treecc” is similar to that used by ”yacc”. A simple rule-based language is devised that is used to describe the intended behaviour declaratively. Embedded code is used to provide the specific implementation details. A translator then converts the input into source code that can be compiled in the usual fashion.*

Treecc

“ *The translator is responsible for generating the tree building and walking code, and for checking that all relevant operations have been implemented on the node types. Functions are provided that make it easier to build and walk the tree data structures from within a ”yacc” grammar and other parts of the compiler.*

Weatherley [2002]

Treecc: a simple example for expressions (1/2)

```
%node expression %abstract %typedef
%node binary expression %abstract = {
  expression* expr1;
  expression* expr2;
}
%node unary expression %abstract = {
  expression* expr;
}
%node intnum expression = {
  int num;
}
%node floatnum expression = {
  float num;
}
%node plus binary
%node minus binary
%node multiply binary
%node divide binary
%node negate unary
```

Treecc: a simple example for expressions (2/2)

```
%union {
  expression* node;
  int inum;   float fnum;
}
%token <inum> INT
%token <fnum> FLOAT
%type <node> expr
%%
expr: INT
    { $$ = intnum_create($1); }
 | FLOAT
    { $$ = floatnum_create($1); }
 | '(' expr ')'
    { $$ = $2; }
 | expr '+' expr
    { $$ = plus_create($1, $3); }
       // ...
 ;
```

Syntax Definition Formalism [visser, 1995]

```
module Tiger-Expressions
imports Tiger-Lexicals Tiger-Literals
exports
  sorts Exp Var
  context-free syntax
    Id                          -> Var        {cons("Var")}
    Var                         -> LValue
    LValue "." Id               -> LValue     {cons("FieldVar")}
    LValue "[" Exp "]"          -> LValue     {cons("Subscript")}
    IntConst                    -> Exp        {cons("Int")}
    StrConst                    -> Exp        {cons("String")}
    "nil"                       -> Exp        {cons("NilExp")}
    LValue                      -> Exp
    Var "(" {Exp ","}* ")"      -> Exp        {cons("Call")}
    Id "=" Exp                  -> InitField  {cons("InitField")}
    TypeId "{" {InitField ","}* "}"     -> Exp  {cons("Record")}
    TypeId "[" {Exp ","}+ "]" "of" Exp -> Exp  {cons("Array")}
```

Part of the Spoofax Language workbench used to develop DSL.

Developing a new languge

Saving and exchanging AST in non-trivial!

BUT hopefully tools exist!

CastXML

C-family abstract syntax tree XML output tool.

-

Protobuff

- Extensible mechanism for serializing data
- Language neutral
- Plateform neutral
- Compact (3 to 10 times smaller than XML)
- Back and Forward compatibility
- Generate data access classes that are easier to use programmatically
- Support for RPC

Protobuf

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }
```

Protobuff

```
// Serialization
Person person;
person.set_name(``John Doe'');
person.set_id(1234);
person.set_email(``jdoe@example.com'');
fstream output(``myfile'', ios::out | ios::binary);
person.SerializeToOstream(&output);

// De-Serialization
fstream input(``myfile'', ios::in | ios::binary);
Person person;
person.ParseFromIstream(&input);
cout << ``Name: `` << person.name() << endl;
cout << ``E-mail: `` << person.email() << endl;
```

Summary