Compiler Techniques MN1
The nano-C Language

February 8, 2005

1 Overview

nano-C is a small subset of C, corresponding to a typical imperative, procedural language. The following sections describe in more detail which language elements are present.

Every nano-C program is also a valid C program. The syntax and semantics of nano-C is the same as that for full C, within the restrictions described here.

2 Lexical elements

- Decimal integer constants.
- Alphanumeric identifiers: non-empty sequences of letters or digits starting with a letter.
- Keywords: else, if, int, return, void, and while.
- Special symbols: :=, !, &&, (,), *, +, (comma), -, /, ;, <=, <, ==, =, >, > [ ], { },{ }, }.

- White space characters: blank (32), newline (10), carriage return (13), form feed (12), and tab (9).
  Comments: /* followed by anything and terminated by */ and // followed by anything and terminated by end of line.

3 Syntax

- Primary expressions: constants, identifiers, function calls, array indexing, and parentheses.
  Unary expressions with the ! and - unary operators.
  Binary expressions with the +, -, *, /, <, >, <=, >=, ==, !=, &&, || and = operators.
- Statements: expression statements, the empty statement, if statements with or without else, while statements, return statements, and compound statements (blocks) { statements }.
Local variable declarations are only permitted at the top-level function body block, not in nested blocks.

- Variable declarations: base type followed by variable name, and for arrays followed by the array size in square brackets.
  The only allowed base type is \texttt{int}.
  Multi-dimensional arrays, pointers, and structures are not included. Arrays cannot be parameters to functions.
  Initialisers in variable declarations are not included.

- Function definitions: return type or \texttt{void}, function name, parameter list, and body (compound statement) in that order.
  The parameter list is either \texttt{void}, meaning no parameters, or a sequence of variable declarations separated by , (comma).
  An external (library) function can be declared by writing a function definition without body, terminated with a ; (semi-colon).
  Variable-arity functions are not included.

4 Program execution

- Execution starts at the user-defined function \texttt{main} which takes no parameters and returns \texttt{int}. Execution ends when \texttt{main} returns.

- The standard library is nano-C-specific since nano-C excludes variable-arity functions, and this makes \texttt{printf} and \texttt{scanf}-like functions impossible. To use the library, include the following declarations at the start of your nano-C source file:

  ```c
  void printChar(int ch); // prints to stdout
  void printInt(int i); // prints to stdout
  int readChar(void); // reads from stdin
  int readInt(void); // reads from stdin
  ```
5 Example

/* This is an example nano-C program. */
void printInt(int i);

int fac(int n)
{
    if (n < 2)
        return n;
    return n * fac(n-1);
}

int a[2];

int sum_a(int n)
{
    int i;
    int s;

    i = 0;
    s = 0;
    while (i < n) {
        s = s + a[i];
        i = i + 1;
    }
    return s;
}

int main(void)
{
    a[0] = fac(5);
    a[1] = 27;
    printInt(sum_a(2)); // prints 147
    return 0;
}
6 Informal grammar for nano-C

This is an informal context-free grammar for nano-C:

- The start symbol is `program`.
- Terminals are written within double-quotes.
- `/empty/` denotes the empty string.
- `int const` and `ident` denote classes of lexical elements.
- Associativity and precedence for expression operators is not expressed.
- The grammar has not been adjusted to fit any particular parsing method.

```plaintext
program ::= topdec_list

topdec_list ::= /empty/ | topdec topdec_list

topdec ::= vardec ";" | fundec

vardec ::= scalardec | arraydec

scalardec ::= typename ident

arraydec ::= typename ident "[" intconst "]"

typename ::= "int"

fundec ::= funtype ident "(" formals ")" funbody

funtype ::= typename | "void"

funbody ::= "{" locals stats "}" | ";;"

formals ::= "void" | formal_list

formal_list ::= scalardec | scalardec "," formal_list

locals ::= /empty/ | vardec ";" locals

stmts ::= /empty/ | stmt stmts

stmt ::= expr ";"

| "return" expr ";" | "return" ";"
| "while" condition stmt
| "if" condition stmt else_part
| "{" stmts "}"
| ";;"

else_part ::= /empty/ | "else" stmt

condition ::= "(" expr ")"

expr ::= int const

| ident | ident "[" expr "]"
| unop expr
| expr binop expr
| ident "(" actuals ")"
| "(" expr ")"

unop ::= "-" | "!"

binop ::= "+" | "-" | "*" | "/"

| "<" | ">" | "<=" | ">=" | "!=" | "=="
| "|" | "&&"
| "=="

actuals ::= /empty/ | expr_list

expr_list ::= expr | expr "," expr_list
```
6.1 Parsing top-level declarations

Top-level declarations have parsing problems with both top-down and bottom-up parsing methods. The problem comes from the fact that when \texttt{int} has been parsed and is followed by \texttt{ident}, the parser cannot determine if the \texttt{int} should be a \texttt{typename} for a variable declaration or a \texttt{functype} for a function declaration.

The first half of the solution is to eliminate \texttt{functype} and move its production into \texttt{topdec}.

The second half of the solution can be done in two different ways.

One alternative is to make two versions of the (old) \texttt{functype} production, one with \texttt{typename} first, and one with \texttt{void} first. This eliminates the ambiguity, but the cost is an additional production.

Another alternative is to eliminate \texttt{functype} and allow \texttt{void} as an alternative in \texttt{typename}. This however means that the syntax tree can contain variable or array declarations with \texttt{void} as base type: they must be detected and rejected, either by the syntax-tree building code, or by the type checking pass.

6.2 Expression operator precedence table

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>primary and postfix expressions</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>\texttt{a[]}</td>
</tr>
<tr>
<td>16</td>
<td>\texttt{f(...)}}</td>
</tr>
<tr>
<td>prefix unary operators</td>
<td>14</td>
</tr>
<tr>
<td>infix operators</td>
<td>13L</td>
</tr>
<tr>
<td>12L</td>
<td>+</td>
</tr>
<tr>
<td>10L</td>
<td>&lt;</td>
</tr>
<tr>
<td>9L</td>
<td>==</td>
</tr>
<tr>
<td>5L</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>4L</td>
<td></td>
</tr>
<tr>
<td>2R</td>
<td>=</td>
</tr>
</tbody>
</table>

The numbers to the left indicate precedence; larger numbers indicate higher precedence. \texttt{L} indicates left-associative operators and \texttt{R} indicates right-associative operators. The table only describes \texttt{C} operators included in nano-C.