04/14/2011

Recitation #12

Administrative

- Final Project (due **April 26 @ 11:59 p.m.; No Lateness allowed**)
- Assignment 4 is graded
  - Dr. Hughes will weight final project more if you get a better grade
  - You can find grading criteria and test / reference files on my Wiki page

Final Project – Rascal Compiler

- Dr. Hughes posted his Flex/Bison reference solution again → That should be immensely helpful
  You can compile the executable parser with the following statements:

  ```
  flex –oRascalRef.lex.c RascalRef.l
  bison -oRascalRef.c RascalRef.y
  ```

  Then simply compile the source file "`RascalRef.c`" into an executable with the IDE of your choice.

- You got lots of test cases (6 correct inputs, 5 erroneous ones). Make use of them! And come up with some tests of your own.
- I made a little skeleton code with the Flex lexical analyzer and a rudimentary parser that is calling the `yylex()` function to get new tokens → available on course website (project folder) or my Wiki page

Review: How to turn grammar into C-code

- The following rules are taken from an online tutorial on how to implement LL(1) parsers in C# : http://www.itu.dk/people/kfl/parsernotes.pdf (I just converted the code to C syntax)
- There are really only a handful of rules that help you in turning grammar rules into C code
- **Rule 0**: A grammar production of the form $A \rightarrow f_1$ has the code:

```
void A()
{
    parse code for f1;
    return;
}
```

- **Rule 1**: A production of the form $A \rightarrow f_1 | \ldots | f_n$ has the code:

```
void A()
{
    switch( currentToken )
    {
        case t11:
        ...
        case t1m:
            parse code for alternative f1;
            return;
```

```
        ...
        case tn1:
        ...
        case tnm:
            parse code for alternative fn;
            return;
        default:
            error("Expected t11 or … or tnm");
            return;
    }
}
```

where $\{t_{i1},...,t_{im}\} = \text{FIRST}(f_i)$ is the first-set of alternative f_i for $i = 1,...,n$. Note that for this to work, all FIRST sets have to be disjoint! See the example below for how we would deal with that, if that's not the case.

- **Rule 2**: A production of the form $A \to f_1 | \ldots | f_n | \varepsilon$ has the same code as rule 1 with the exception of the "default" case for the switch statement:

```
void A()
{
    switch( currentToken )
    {
        case t11:
        ...
        case t1m:
            parse code for alternative f1;
            return;
        ...
        case tn1:
        ...
        case tnm:
            parse code for alternative fn;
            return;
        default:
            return;
    }
}
```

- **Sequence**: The parse code for an alternative f which is a sequence $e_1\ e_2 \ldots e_m$ is

  $P(e_1)$

  …

  $P(e_m)$

  where the parse code $P(e_i)$ for each symbol is defined below. Note that when the sequence is empty, i.e. $m = 0$, the parse code is empty too.

- **Nonterminal**: The parse code $P(A)$ for a non-terminal A is a call `A()` to its parsing method.
- **Terminal**: The parse code $P(\text{"}c\text{"})$ for a terminal "c" depends on its position e_j in the sequence $e_1\ e_2 \ldots e_m$.

  If the terminal is not the first symbol e_1, then we must check that c is the current token, and, if so, read the next token:

```
if (currentToken != c)
{
    error("Expected token c");
    return;
}
getNextToken();
```

If the terminal is the first symbol e_1, then this check has already been made by the `switch` code for alternatives, so we just need to read the next token:

```
getNextToken();
```

Example of turning grammar rule into C-Code

Great case in point: Write a function for the non-terminal simple.statement:

```
simple.statement:
       assignment.statement |
       procedure.call |
       /* empty */
       ;

assignment.statement:
       IDENT sub assign expression

procedure.call:
       IDENT  LPAREN call.list RPAREN  |
       IDENT

sub:
       LBRACKET expression RBRACKET |
       /* empty */
       ;
```

Problem with writing code for simple.statement is, that FIRST sets of both assignment.statement and procedure.call are not disjoint, i.e. FIRST(assignment.statement) = FIRST(procedure.call) = { IDENT }.

```
void simple_statement
{
    switch(currentToken)
    {
        case IDENT: ???
    }
}
```

Since we already know that both alternatives in simple.statement start out with the terminal IDENT, why don't we express that in our grammar. In a way, we are doing a kind left factoring and introduce a new non-terminal A.

```
simple.statement:
       IDENT A | /* empty */ ;
```

```
A:
       assignment.statement | procedure.call;

assignment.statement:
       sub assign expression

procedure.call:
       LPAREN call.list RPAREN  |
       /* empty */;
```

Let's look at the FIRST sets now:
FIRST(assignment.statement) = FIRST(sub assign expression) = { LBRACKET } ∪ FIRST(assign) = { LBRACKET, EQ, ASSIGN }
FIRST(procedure.call) = { LPAREN, epsilon }

Great, the FIRST sets are now disjoint and we can write the following code for simple.statement and A:

```
void simple_statement
{
    if (currentToken != IDENT)
        return;
    getNextToken();
    A();
}

void A()
{
    switch(currentToken)
    {
        case LBRACKET:
        case EQ:
        case ASSIGN:
            assignment_statement();
            return;

        case LPAREN:
        default:
            procedure_call();
            return;
    }
}
```

The lesson that you should draw from this example is that you shouldn't hesitate to modify the original grammar if it eases the parsing process. Don't try to customize your functions to fit the problem, but modify the problem if possible. That way you can always refer back to the few rules above and apply them to build your recursive descent parser.

Expression Handling

- Expressions show up everywhere in the grammar for the final project, so please make sure that you tackle them early on
- The following grammar rules need to be implemented:

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T / F \mid F$$
$$F \rightarrow (E) \mid NUMBER$$

- According to the rules above we would end up going into an infinite loop calling `E()` over and over again without advancing the token → due to left recursion
- One possible solution to the problem is to remove left recursion (see notes for lab 8) and apply the rules we just learned; if you are interested in that route, please take a look at http://www.itu.dk/people/kfl/parsernotes.pdf, because they specifically tackle this case in their notes
- Probably the more elegant way was introduced by Dr. Hughes:

```
int E()
{
    int op1, op2; char op;
    op1 = T();
    while ((currentToken == PLUS) || (currentToken == MINUS))
    {
        if (currentToken == PLUS) op = '+';
        else op = '-';

        getNextToken();
        op2 = T();
        op1 = emit(op, op1, op2);
    }
    return op1;
}

int T()
{
    int op1, op2;
    op1 = F();
    while ((currentToken == TIMES) || (currentToken == DIVIDE))
    {
        if (currentToken == TIMES) op = '*';
        else op = '/';

        getNextToken();
        op2 = F();
        op1 = emit(op, op1, op2);
    }
    return op1;
}

int F()
{
    int op1;
    switch (currentToken)
```

```
    {
        case LPAREN :
            getNextToken();
            op1 = E();
            if (currentToken != RPAREN) error();
            break;
        case INTEGER :
            op1 = emit("CON", yylval, 0);
            break;
        default:
            error();
            break;
    }

    getNextToken();
    return op1;
}
```

More comments

- **Casting** has highest precedence
  - e.g. in the expression
    `{unit} a * b`
    only `a` is cast to `unit`, before it is multiplied with `b`
  - If you want to cast the result of an operation, simply put parentheses around the expression, e.g.
    `{unit} (a * b)`
    where first the multiplication of `a` and `b` is calculated, before the result is cast to `unit`
- Dedicated slot PROGRAMSLOT in symbol table for main program, so interpreter knows where to start execution (because first line of intermediate code might be from a procedure)
- **Symbol table** handling
  - Units and aliases are stored in symbol table
  - Duplicate symbol table entries might be found, so only enter new variable if it is at different nesting level, e.g.
    `if (symtab[$1.ival].nesting != nesting) store_...()`
  - For aliases, symbol table entry `.unit` points to canonical unit entry

- **Unions** in C code
  - Unions allow one same portion of memory to be accessed as different data types, since all of them are in fact the same location in memory
  - All the elements of the union declaration occupy the same physical space in memory
  - Its size is the one of the greatest element of the declaration
  - Dr. Hughes uses that for the symbol table:

```
union symtab_entry
{
   struct proc_type  proc;
   struct unit_type  unit;
   struct var_type   var;
   struct const_type cons;
};
struct symtab_type
{
   char name[IDLENGTH];
   int  nesting;
   int  type;
   int  unit;
   union symtab_entry sym;
} ;

struct  symtab_type symtab[SYM_MAX];
```

  - Each element of the symbol table is either a proc_type, unit_type, var_type, or const_type
  - You differentiate them by how you access a symtab entry, e.g. `symtab[symsize].sym.proc.base = triple;`
  - Always use the same accessor!
  - Other cool use: access unite elementary type with array or structures of smaller elements:

```
union mix_t
{
    long l;
    struct
    {
        short hi;
        short lo;
    } s;
    char c[4];
} mix;
```