Introduction to Compilers, Part 2

ITP 435
Week 10, Lecture 2

Parser (Syntax Analysis)

- Given a series of tokens from the lexer/scanner, two main jobs:

1. Analyze the syntax of the source code, make sure that it conforms to the language.

2. Generate an IR – in our case, an AST (Abstract Syntax Tree).

Yacc/Bison

- Yacc - Yet Another Compiler Compiler
- First released in 1970

- Given a *context-free grammar*, it generates a parser that can parse said grammar

- Bison is the open source version of Yacc that we will be using

Context-Free Grammar

- The grammar defines the rules of where tokens have to be placed in order to create a valid statement/expression
- It's context-free because there's no sense of context (so words don't have different meanings depending on where they are used)
- The simplest way to express grammars is using *Backus-Naur Form* (BNF)

Backus-Naur Form (original syntax)

- Simple example for addition/subtraction:

```
<expression> ::== <expression> "+" <expression>
                | <expression> "-" <expression>
                | <number>
```

- Meta-symbols:
- `::==` "Is defined as"
- | "or"
- <> To signify the name of a grammar definition

Left Recursion

- Left recursion means that the leftmost component of a grammar definition is that definition itself

- Eg. This is left recursive:

```
<expression> ::== <expression> "+" <expression>
```

- Not all parser-generators support left recursion. But Yacc/Bison does, and it makes life so much easier.

Full Math Grammar

- The grammar I used for my simple calculator was as follows:

```
<expression> ::== <expression> "+" <expression>
                | <expression> "-" <expression>
                | <expression> "*" <expression>
                | <expression> "/" <expression>
                | "(" <expression> ")"
                | <integer>
```

Where `<integer>` is defined in the lexer as:

`[0-9]+`

LALR (Look-Ahead LR) Parsing

- Given the following math expression:

```
5 + 10
```

- An LR-family parser, like Bison, would parse it in this order:

```
integer (5)
Expression (integer)
integer (10)
Expression (integer)
addition expression
```

More Complex Grammar Example

- Function call expression in C (ignoring function pointers):

```
<function_call> ::== <identifier> "(" ")"
                   | <identifier> "(" <params> ")"

<params> ::== <expression>
            | <params> "," <expression>
```

Params More Closely

- What would fit this params rule?

```
<params> ::== <expression>
            | <params> "," <expression>
```

- Well…

```
<expression>
<expression> "," <expression>
<expression> "," <expression> "," <expression>
...
```

A Sample Question

Write a BNF grammar for a list of one or more C-style declaration statements with the following restrictions:

- The type can only be `int` or `float`
- `<id>` is the identifier (NOTE: You don't need to write regex for this)
- It optionally is assigned to a `<num>` token (again, don't write the regex for this)
- It ends with a semi-colon

So for example, this would be a valid list of declaration statements:

```
int var1 = 5;
float varxxx;
float yyy = 20;
```

Sample Question, Cont'd

- First, we can represent the "type" grammar rule as follows:

```
<type> ::== "int"
          | "float"
```

Sample Question, Cont'd

- Next, we can define the declaration statement as follows:

```
<decl> ::== <type> <id> "=" <num> ";"
          | <type> <id> ";"
```

Sample Question, Cont'd

- Finally, a list of declarations:

```
<decl_list> ::== <decl>
              | <decl_list> <decl>
```

In-class Activity

Calc Full Example

- Look at full calc example which does calculations:

- https://github.com/chalonverse/CalcSamples

Bison Grammar

- A grammar in Bison looks very similar to BNF

- Here's the calculator grammar in Bison:

```
expr : expr TPLUS expr
     | expr TMINUS expr
     | expr TMULT expr
     | expr TDIV expr
     | TLPAREN expr TRPAREN
     | TINTEGER
;
```

- (Where TPLUS, TMULT, etc. are in our token enum)

Unions in C/C++

- A union is like a C-style struct (eg. Plain old data), except each member of the union occupies the same memory space.

- Eg. the `sizeof(Vector3)` below is 12:

```
union Vector3
{
    struct { float x, y, z; }; // 3 * 4 = 12
    float val[3]; // 3 * 4 = 12
};
```

Unions in C/C++, Continued

- I can then access any member of the union by either naming convention:

```
Vector3 temp;
// The below expression is always true,
// because temp.x and temp.val[0]
// are the same element in memory
temp.x == temp.val[0]
```

- Bison converts the %union statement to a C-style union

Tokens in Bison

- There are two parts to defining a token or grammar definition.

- The union statement specifies which data type(s) you could potentially store for each grammar rule that is matched
- The following:

```
%union {
    std::string* string;
    int token;
}
```

- Means that for each grammar rule match, I will either be storing an integer or a pointer to a string

Tokens in Bison, Cont’d

- Once the %union is defined, then you can list out all of the tokens (also known as terminal symbols)

- For each token, you have to say which member of the union they use:

```
%token <string> TINTEGER
%token <token> TPLUS TMINUS TMULT TDIV TLPAREN TRPAREN
```

Tokens in Flex

- Once the tokens are defined in Bison, we then need to return the appropriate token values in Flex:

```
[ \t\n]  { }
[0-9]+   { SAVE_TOKEN; return TINTEGER; }

"+"      { return TOKEN(TPLUS); }
"-"      { return TOKEN(TMINUS); }
"*"      { return TOKEN(TMULT); }
"/"      { return TOKEN(TDIV); }
"("      { return TOKEN(TLPAREN); }
")"      { return TOKEN(TRPAREN); }
```

SAVE_TOKEN/TOKEN

- They are just macros that were defined in the top of the Flex source file, to make things easier:

```
#define SAVE_TOKEN calclval.string = new std::string(yytext, yyleng)
#define TOKEN(t) (calclval.token = t)
```

- Notice that they are storing the data in the union

Operator Precedence

- To define operator precedence in Bison, you can use %left for left-to-right operators and %right for right-to-left operators:

```
%left TPLUS TMINUS
%left TMULT TDIV
```

- The first operators listed have the lowest precedence, and the latter ones have the highest

Grammar Definitions in Bison

- Just like how you have to define all the tokens, you have to also define all the grammar definitions:

```
%type <token> expr
```

- (Normally, we would have a type specific to a node in our AST that would be in the union).

Grammar Actions

- The power of Bison is that you can give it *actions* (in C/C++) to perform when a grammar match is made.
- Actions can access any element within the grammar matching pattern, by using $1 to access the first, $2 to access the second, so on.
- For instance:

```
/* Compute is a global function made for this example */
expr TPLUS expr { std::cout << "+" << std::endl; Compute($2); }
```

Grammar Actions, Cont'd

- The type of $1 and so on directly matches the types defined earlier with %token or %type statements.

- So given:

```
%token <token> TPLUS TMINUS TMULT TDIV TLPAREN TRPAREN
```

- That means the type of $2 below will be an `int`, because in the %union, one of the members was an int called token.

```
expr TPLUS expr { Compute($2); }
```

Calc with AST

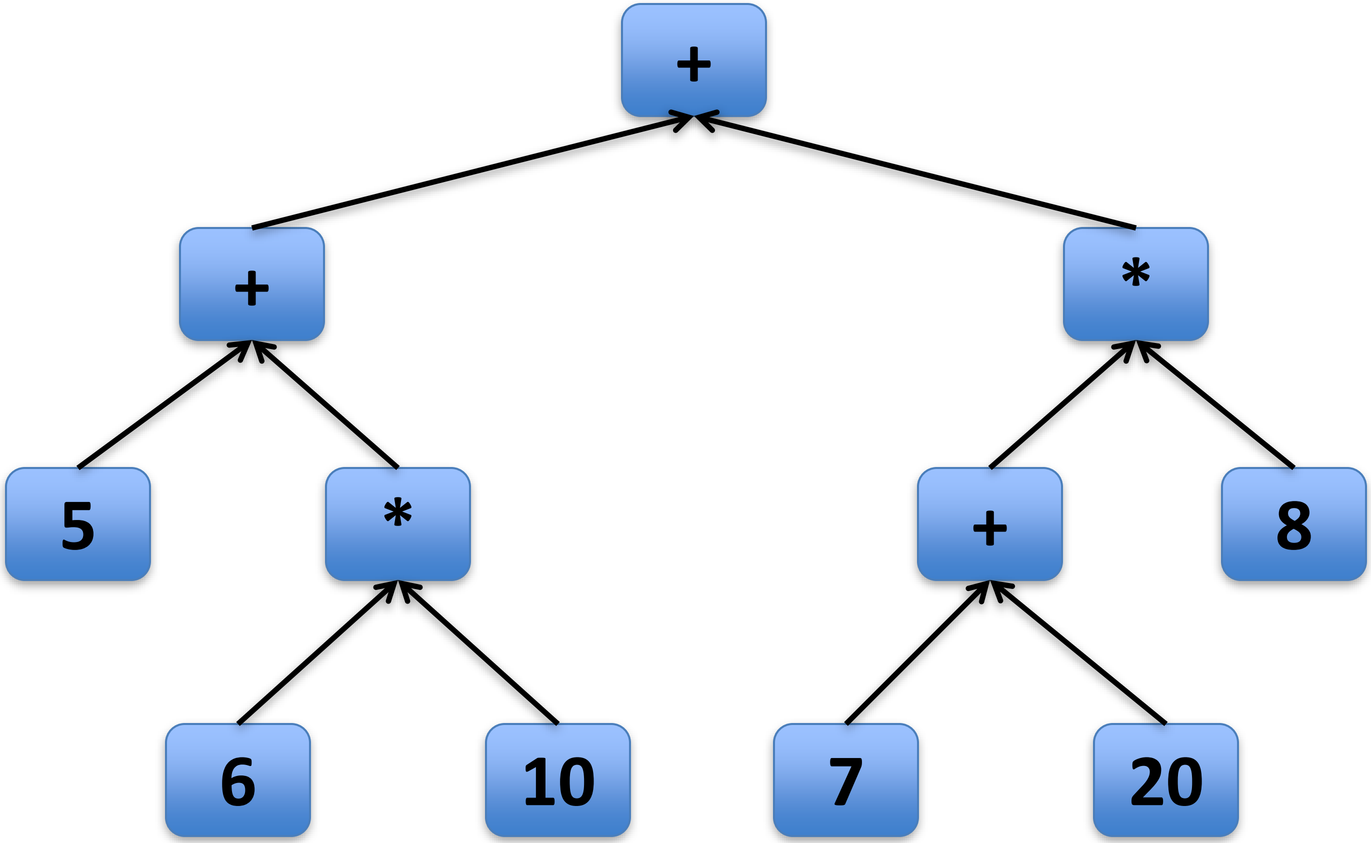
- Suppose the input is:

```
5+6*10+(7+20)*8
```

- As demonstrated in calc, we can instantly calculate this.

- But what if we wanted to generate an AST so we could then do further operations (such as generate assembly)?

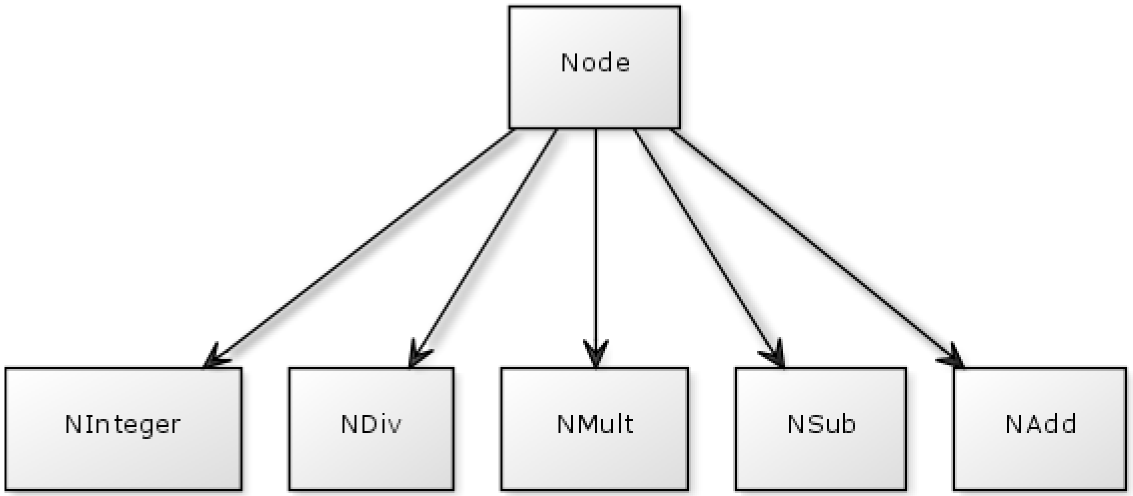
AST for: 5+6*10+(7+20)*8



Post-Order Traversal: 5,6,10,*,+,7,20,+,8,*,+

Using an AST with Bison

- First, create a class hierarchy for all nodes in the AST:



- In this case, it's a very shallow hierarchy. But it isn't always.

Using an AST with Bison, Cont'd

- Next, declare your classes in a header, as appropriate. Eg:

```
class Node
{
public:
   virtual void debugOut() = 0;
};
class NAdd : public Node
{
public:
   NAdd(Node* lhs, Node* rhs);
   void debugOut();
private:
   Node* m_lhs;
   Node* m_rhs;
};
```

Using an AST with Bison, Cont'd

- Next, add pointers to abstract classes to the %union definition:

```
%union {
    Node* node;
    std::string* string;
    int token;
}
```

- No concrete classes, because Bison %type definitions have no concept of inheritance, since it generates C code.

Using an AST with Bison, Cont'd

- Next, update the %type definition as appropriate. For count, it's simple:

```
%type <node> expr
```

Using an AST with Bison, Cont'd

- $$ in Bison refers to the value stored for that particular match.
- We simply need to construct the correct classes and store them in $$ in our actions.

```
expr : expr TPLUS expr { $$ = new NAdd($1, $3); }
     | expr TMINUS expr { $$ = new NSub($1, $3); }
     | expr TMULT expr { $$ = new NMult($1, $3); }
     | expr TDIV expr { $$ = new NDiv($1, $3); }
     | TLPAREN expr TRPAREN { $$ = $2; }
     | TINTEGER { $$ = new NInteger(*($1)); }
;
```

Using AST with Bison, Cont'd

- Now we have an AST…but how do we get a pointer to the root node?

- Solution: Add a top-level rule to our grammar, such as:

```
<result> ::== "result" "=" <expression>
```

- Then, add a global pointer to a Node, and simply assign it in the action for that rule in Bison:

```
result : TRESULT TEQUALS expr { g_Result = $3; }
;
```

Calc2 Demonstration

- Let's take a closer look at the calc2 project…