Honors Compilers, G22.3130
Fall 2007: Parser Assignment

Due Monday 10.30.07

1 General Description

Your next task is to construct a parser using a parser generator (e.g. Bison, ML-YACC, etc.) that recognizes strings in the language whose grammar is provided in Section 2 and uses the lexer that you have already written. The parser should create an abstract syntax tree (AST) whose leaves are the necessary terminals (represented at this point by lexeme-token pairs) and whose internal nodes are labeled with the kind of sentence represented by the subtree.

You need to define, at this point, the structure of the abstract syntax tree based on the grammar given at the bottom of this document.

1.1 C++ Representation

For example, the C++ declaration might look like

```
enum node_c { PROGRAM, BLOCK, TYPE_DEF, VAR_DEF, ... }

class Node {
public:
   ...
   node_c get_node_class();
   ...
protected:
   node_c node_class;
   ... }

class Program: public Node {
public:
     Program(...) { node_class = PROGRAM; ... }
     ...
protected:
    token_pair name;
```

```
    node *type_defs[];
    node *var_defs[];
    node *subprog_defs[];
    node *body;
    ...
}
...
```

As you add phases to the compiler, you will need to go back and add new fields to the various node types in order to hold other information (type, size, etc.) that is gathered by subsequent phases. For now, though, it is sufficient to define only those fields required to represent the AST.

1.2 A C Representation

A possible representation in C could be the following:

```
typedef union {
   int inttype;
   struct node *nodetype;
   struct string *strtype;
} value;

/* Define node */
typedef struct node{
   short int kind;
   value left,right;
} node_rec, *node_ptr;
```

and then define a generic function for allocation of a new node

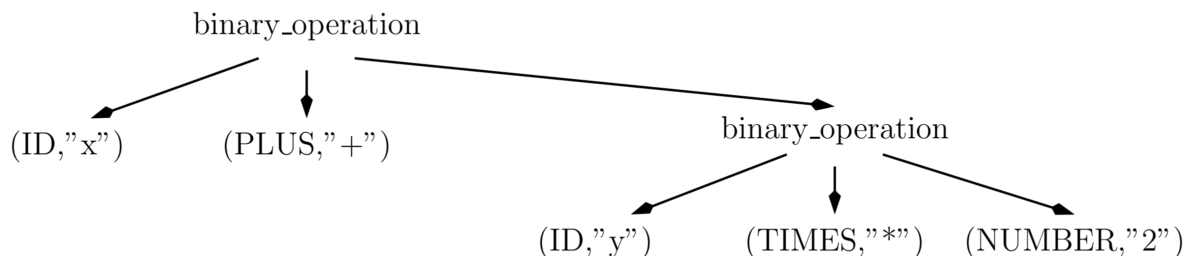
```
node_ptr new_node(int kind, node_ptr left, node_ptr right)
```

which can be used to implement allocation of nodes for integer and string as follows:

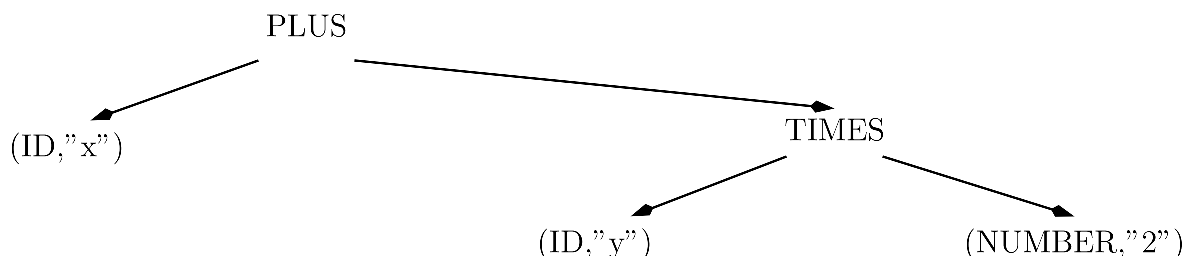
```
node_ptr new_NUMBER_node(int num)
{
   return new_node(NUMBER, (node_ptr) num, NIL);
}
node_ptr new_string_node (char *s)
{
   return new_node(STRING, (node_ptr) s, NIL);
}
```

In case the node should have more than two descendants, such as a function call with a list of parameters, it will be necessary to break it into a list of linked nodes.

Note that there doesn't have to be an interior node in the AST for every non- terminal in the grammar below. For example, the syntax of arithmetic expressions has a large number of non-terminals (such as simple_expression, term, and factor) that are solely introduced for the purpose of specifying operator precedence. There is no need to label interior nodes as being simple_expressions, terms, or factors. For example, $x + y * 2$ can simply be represented as:



where ID, PLUS, etc. are the tokens returned by your lexer. Also, not every terminal in the language needs to appear as a leaf in the AST. For example, there's no need to have leaves labeled with keyword tokens, such as begin, end, if, then, etc., in the AST. Following is an alternative representation of the AST where the terminal + and * appear as the kind of the node:



1.3 The Deliverables

You should turn in:

1. The parser-generator specification along with an updated (if necessary) lexer, and
2. A driver program that reads a program (such as the test programs on the web page) and prints out the AST.
3. The output produced by running your parser on the test programs.

2 The Grammar

The grammar is given in Extended Backus-Naur Form (EBNF). The meaning of the meta-symbols are as follows

MetaSymbol	Meaning
::=	is defined to be
|	alternatively
[X]	0 or 1 instance of X
{X}	0 or more instances of X
(X | Y)	a grouping: either X or Y
( ) [ ]	the terminal symbols ( ) [ ].
xyz	The terminal symbol **xyz**
MetaIdentifier	The non-terminal symbol *MetaIdentifier*

The grammar below is for a subset of Pascal, taken (and revised) from the Pascal User Manual and Report, 3rd edition, by K. Jensen & N. Wirth (Springer, 1985). The language has been substantially simplified by not allowing nested procedures.

The terminal symbols ID, INT, and STRING refer to the identifier and integer literal tokens returned by the lexer.

Program ::= **program** ID ; [*TypeDefinitions*] [*VariableDeclarations*]
[*SubprogramDeclarations*] *CompoundStatement* **.**

TypeDefinitions ::= **type** *TypeDefinition* ; {*TypeDefinition* ; }

VariableDeclarations ::= **var** *VariableDeclaration* ; {*VariableDeclaration* ; }

SubprogramDeclarations ::= {(*ProcedureDeclaration* | *FunctionDeclaration*) ; }

TypeDefinition ::= ID = *Type*

VariableDeclaration ::= *IdentifierList* : *Type*

ProcedureDeclaration ::= **procedure** ID (*FormalParameterList*) ; (*Block* | **forward**)

FunctionDeclaration ::= **function** ID (*FormalParameterList*) : *ResultType* ;
(*Block* | **forward**)

FormalParameterList ::= [*IdentifierList* : *Type* { ; *IdentifierList* : *Type* }]

Block ::= [*VariableDeclarations*] *CompoundStatement*

CompoundStatement ::= **begin** *StatementSequence* **end**

StatementSequence ::= *Statement* { ; *Statement* }

Statement ::= *SimpleStatement* | *StructuredStatement*

SimpleStatement ::= [(*AssignmentStatement* | *ProcedureStatement*)]

AssignmentStatement ::= *Variable* **:=** *Expression*

ProcedureStatement ::= ID (*ActualParameterList*)

StructuredStatement ::= *CompoundStatement*
| **if** *Expression* **then** *Statement* [**else** *Statement*]
| **while** *Expression* **do** *Statement*
| **for** ID **:=** *Expression* **to** *Expression* **do** *Statement*

Type ::= *BasicType* | ID | **array** [*constant* **..***constant*] **of** *Type*
| **record** *FieldList* **end**

ResultType ::= *BasicType* | ID

BasicType ::= **integer** | **string** | **boolean**

FieldList ::= [*IdentifierList* : *Type* { ; *IdentifierList* : *Type* }]

Constant ::= [*sign*] INT

Expression ::= *SimpleExpression* [*RelationalOp* *SimpleExpression*]

RelationalOp ::= < | <= | > | >= | <> | =

SimpleExpression ::= [*sign*] *Term* { *AddOp* *Term* }

AddOp ::= + | **-** | **or**

Term ::= *Factor* { *MulOp* *Factor* }

MulOp ::= * | **div** | **mod** | **and**

Factor ::= INT | STRING | *Variable* |
FunctionReference | **not** *Factor* | (*Expression*)

FunctionReference ::= ID (*ActualParameterList*)

Variable ::= ID *ComponentSelection*

ComponentSelection ::= [(**.** ID *ComponentSelection* | [expression] *ComponentSelection*)]

ActualParameterList ::= [*Expression* { , *Expression* }]

IdentifierList ::= ID { , ID }

Sign ::= + | **-**

In a conditional, an **else** is associated with the nearest **if**... **then**.