Part I

Project advice
Announcements

- HW02 and PR02 submissions are due no later than 5:00 PM on Monday March 12.
- HW01 grades will be available some time next week.
Plan your symbol table implementation well:

2.7.1 Symbol Table Per Scope
(page 86)
How do rules for function definitions work?

**type** identifier `::` pblock `->` identifier  
*Function type.*

**function** identifier₁ `::` identifier₂ sblock  
*Function definition.*
identifier₁ is function name,
identifier₂ is function type
How do functions work?

A function maps a value from its domain to a value in its range:
Project Notes

In C code:

\[
double f(int x) \{ \ldots \} \]

This function's type, int -> double, is anonymous: it has not given an explicit name.
Back to our language:

- **type** identifier ‘::’ pblock ‘->’ identifier  
  Function type.

- **function** identifier₁ ‘::’ identifier₂ sblock  
  Function definition.  
  identifier₁ is function name,  
  identifier₂ is function type

Diagram:
- Domain: \( x \) → \( f(x) \)
- Range: \( f(x) \)
Project Notes

In code:

type foo : (int x) -> double

function f : foo { ... }
How do declarations work?

sblock is:

```
{ ( dblock ) statement-list }
```

dblock is:

```
[ declaration-list ]
```

declaration-list is:

```
declaration ;' declaration-list | declaration
```

declaration is:

```
identifier :' identifier-list
```

LHS is type, RHS is list of variable names.

identifier-list is:

```
identifier ( assignOp constant ) ,' identifier-list | identifier ( assignOp constant )
```
## Example: declaration grammar (figure 5.8)

**Declaration is:**
- `identifier ‘;’ identifier-list`

**Identifier-list is:**
- `identifier ( assignOp constant ) ‘;’ identifier-list | identifier`

**Production** | **Semantic Rules** |
--- | --- |
1 | `D -> T L` | `L.inh = T.type` |
2 | `T -> int` | `T.type = integer` |
3 | `T -> float` | `T.type = float` |
4 | `L -> L₁ , id` | `L₁.inh = L.inh
addType(id.entry, L.inh)` |
5 | `L -> id` | `addType(id.entry, L.inh)` |

*LHS is type, RHS is list of variable names.*
How do declarations work?

{ 
  [ integer : a := 3, b ; 
    real : x, y := 0.2, z := -5.1 ]

  ... statements ... 
}

What is permitted as an initializer?
Project Notes

How do declarations work?

{ [ integer : a := 3, b ;
   real : x, y := 0.2, z := -5.1 ]

... statements ...

Why only constants as initializers?

How do nested scopes work in e.g. Java?
A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions" [p. 304]
Syntax-Directed Definitions

"A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions" [p. 304]

See the 'union' type in the bison docs.
"A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions" [p. 304]
%union{
   struct Basic basic;
   struct ConstantValue k;
   struct ExpressionTypeInfo t;
   struct StatementInfo stmt;
   ... etc ...
}

%type <basic> ID
%type <basic> typeName
%type <basic> primitiveTypeName
%type <basic> T_INTEGER
%type <basic> T_BOOLEAN
%type <basic> T_REAL
%type <basic> T_CHARACTER
%type <basic> T_STRING
%type <basic> parameter_list
%type <basic> non_empty_parameter_list
%type <basic> parameter_declaration

%type <k> C_INTEGER
%type <k> C_REAL
%type <k> C_CHARACTER
%type <k> C_STRING
%type <k> C_TRUE
%type <k> C_FALSE
%type <k> constant
%type <k> initialization

%type <t> assignable
%type <t> expression
%type <t> expression1

etc.
struct ConstantValue {
    struct SymbolTableEntry * actualType;
    int lineNo;
    int colNo;
    enum ConstantType type;
    union {
        int i;
        double r;
        bool b;
        char c;
        char * s;
    } value;
    struct InstructionList * trueList;
    struct InstructionList * falseList;
};

struct Basic {
    int lineNo;
    int colNo;
    char * s;
};

struct ExpressionTypeInfo {
    struct SymbolTableEntry * actualType;
    struct SymbolTableEntry * expectedType;
    struct InstructionList * trueList;
    struct InstructionList * falseList;
    struct SymbolTableEntry * addr;
};

struct StatementInfo {
    int next; /* instruction following this statement */
};
declaration : typeName COLON {
    $<t>$.actualType = lookup(symbolTable, $1.s);
    if ($<t>$.actualType == NULL) {
        $<t>$.actualType = undefType;
        yyerror( . . . );
    }
}

identifier_list ;

initialization : assignOp constant {
    $$ = $2;
    typeCheck( ($<t>-1).actualType, ($2).actualType );
}
    | { $$ .type = NOT_PRESENT; }

identifier_list :
    ID initialization COMMA {
    insertLocalVariableInSymbolTable($1.s, $<t>0.actualType, &$2);
    $<t>$ = $<t>0;
}
    identifier_list |
    ID initialization {
    insertLocalVariableInSymbolTable($1.s, $<t>0.actualType, &$2);
    $<t>$ = $<t>0;
} ;
The C code in an action can refer to the semantic values of the components matched by the rule with the construct $n$, which stands for the value of the $n$th component. The semantic value for the grouping being constructed is $$.

[...]

The mid-rule action can also have a semantic value. The action can set its value with an assignment to $$, and actions later in the rule can refer to the value using $n$. Since there is no symbol to name the action, there is no way to declare a data type for the value in advance, so you must use the `\$<...>\' construct to specify a data type each time you refer to this value.

http://dinosaur.compilertools.net/bison/bison_6.html#SEC46
rules
(examples)

declaration
  : typeName
      COLON
      {
        <$t>$.actualType = lookup(symbolTable, $1.s);
        if ($<t>$.actualType == NULL) {
          <$t>$.actualType = undefType;
          yyerror( . . . );
        }
      }
  identifier_list
  ;

initialization
  : assignOp
      constant
      {
        $$ = $2;
        typeCheck( ($<t>-1).actualType, ($2).actualType );
      }
      | {
        $$\.type = NOT_PRESENT; }
  ;

$n with n zero or negative is allowed for reference to tokens and
groupings on the stack before those that match the current rule. This is a
very risky practice, and to use it reliably you must be certain of the
context in which the rule is applied. Here is a case in which you can use
this reliably:

 foo:      expr bar '+' expr  { ... }
        | expr bar '-' expr  { ... }
        ;

 bar:      /* empty */
        { previous_expr = $0; }
        ;

As long as bar is used only in the fashion shown here, $0 always refers to
the expr which precedes bar in the definition of foo.

http://dinosaur.compilertools.net/bison/bison_6.html#SEC46
Work incrementally #1

Ensure basic ST functionality works

- add entry for identifier
- look up identifier in local table (in declarations)
- look up identifier in chain of tables (in statements)
Work incrementally #2

- handle basic declaration:
  \[ \text{integer} : x \]

- handle declaration list:
  \[ \text{integer} : x ; \text{real} : y \]

- handle multiple declarations:
  \[ \text{integer} : x, y \]

- handle declarations with initializers:
  \[ \text{integer} : x := 5, y \]
Work incrementally #3

- generate undeclared identifiers:
  \[
  \{ x := x; \} \\
  \{ [\text{foo} : x] x := x; \}
  \]

- generate error for duplicate declarations:
  \[
  [\text{integer} : x; \text{real} : x]
  \]

- add error tokens to strategic places in grammar rules
Other points to consider

- command-line arguments

```c
#include <stdio.h>

int main(int argc, char * argv[]) {
    for (int i=0; i<argc; i++) {
        printf("program argument %d is %s\n",i,argv[i]);
    }
    return 0;
}
```

- Is lexer or parser better able to produce ASC?

- When is it easiest to produce ST output?
Symbol table entries

What does an entry in your symbol table look like?
Symbol table entries

What information do you need to store for each of the following?

- variable
- array type
- record type
- function type
- function definition
- primitive type

Refine over time
Symbol table structure

enum DeclarationType { PRIMITIVE_TYPE_DECLARATION, ARRAY_TYPE_DECLARATION, RECORD_TYPE_DECLARATION, FUNCTION_TYPE_DECLARATION, FUNCTION_DEFINITION, VARIABLE_DECLARATION };

struct SymbolTableEntry {
  char * name;
  enum DeclarationType variant;
  union {
    struct PrimitiveType pt;
    struct ArrayType at;
    struct RecordType rt;
    struct FunctionType ft;
    struct FunctionDefinition fd;
    struct VariableDeclaration vd;
  } entry;
};


Symbol table structure

```c
enum DeclarationType { PRIMITIVE_TYPE_DECLARATION,
ARRAY_TYPE_DECLARATION, RECORD_TYPE_DECLARATION,
FUNCTION_TYPE_DECLARATION, FUNCTION_DEFINITION,
VARIABLE_DECLARATION };

struct SymbolTableEntry {

    char * name;
    enum DeclarationType variant;

    union {
        struct PrimitiveType pt;
        struct ArrayType at;
        struct RecordType rt;
        struct FunctionType ft;
        struct FunctionDefinition fd;
        struct VariableDeclaration vd;
    } entry;

};
```
struct PrimitiveType {
    int    size; // in bytes
};

struct ArrayType {
    int    size;
    int    dimensions;
    struct VariableDeclaration declaration;
};

struct RecordType {
    int    size;
    struct SymbolTable * declaration_list // use a symbol table!
};

struct FunctionType {
    int    size;
    struct SymbolTable * domainType; // use a symbol table!
    struct SymbolTableEntry * rangeType;
};

struct FunctionDefinition {
    struct SymbolTableEntry * type; // this will be a FunctionType
    int    codePointer; // type may not be correct
};

struct VariableDeclaration {
    int    size;    // size == |primitive type|, |pointer| for array, record, function
    int    offset;  // used to determine layout of records, and stack frames
    struct SymbolTableEntry * type;
    bool   parameter; // true --> parameter, false --> local (non-parameter)
    bool   hasInitialization;
    struct ConstantValue * initializationValue;
};
Using union

```c
void addEntryToSymbolTable(struct SymbolTable * st,
                           struct SymbolTableEntry * e) {
  if (FLAG_st != 0) {
    printf("%16s : %5d : %16s : %10s \n",
           getName(e), getNumber(st), getType(e), getAnnotation(e));
  }
  st->entries = newSymbolTableEntryNode(e, st->entries);
}
```

Just some sample code - don't get hung up on the specifics!
void addEntryToSymbolTable(struct SymbolTable * st,  
    struct SymbolTableEntry * e) {  
    if (FLAG_st != 0) {  
        printf("%16s : %5d : %16s : %10s \n",  
            getName(e), getNumber(st), \textbf{getType(e)}, getAnnotation(e));  
    }  
    st->entries = newSymbolTableEntryNode(e, st->entries);  
}
char * getType(struct SymbolTableEntry * e) {
    if (e == NULL) {
        printf("** internal compiler error: getting type of NULL SymbolTableEntry");
        return "** NULL";
    }

switch (e->variant) {
    case PRIMITIVE_TYPE_DECLARATION:
        return e->name;
    case ARRAY_TYPE_DECLARATION: {
        char * domainType = arrayDomain2string(e->entry.at.dimensions);
        char * map = " -> ";
        char * rangeType = e->entry.at.declaration.type->name;
        char * answer = (char *) malloc((strlen(domainType)+strlen(map)+strlen(rangeType)+1) * sizeof(*answer));
        answer[0] = '\0';
        sprintf(answer, "%s%s%s", domainType, map, rangeType);
        return answer; }
    case RECORD_TYPE_DECLARATION: {
        char * answer = cartesianProduct2string(e->entry.rt.declaration_list);
        return answer; }
    case FUNCTION_TYPE_DECLARATION: {
        char * domainType = cartesianProduct2string(e->entry.ft.domainType);
        char * map = " -> ";
        char * rangeType = e->entry.ft.rangeType->name;
        char * answer = (char *) malloc((strlen(domainType)+strlen(map)+strlen(rangeType)+1) * sizeof(*answer));
        answer[0] = '\0';
        sprintf(answer, "%s%s%s", domainType, map, rangeType);
        return answer; }
    case FUNCTION_DEFINITION:
        return e->entry.fd.type->name;
    case VARIABLE_DECLARATION:
        return e->entry.vd.type->name;
    default:
        printf("** internal compiler error: illegal variant used in SymbolTableEntry");
        return "** NULL";
}
}
constructing SymbolTableEntries

struct SymbolTableEntry * newPrimitiveEntry(char * id, int size);

struct SymbolTableEntry * newArrayEntry(char * id, int dimensions, struct SymbolTableEntry * type, struct ConstantValue * initialization);

struct SymbolTableEntry * newRecordEntry(char * id);

struct SymbolTableEntry * newFunctionTypeEntry(char * id, struct SymbolTable * domainType, struct SymbolTableEntry * rangeType);

struct SymbolTableEntry * newFunctionDefinitionEntry(char * id, struct SymbolTableEntry * type, int codePointer);

struct SymbolTableEntry * newLocalEntry(char * id, struct SymbolTableEntry * type, struct ConstantValue * initialization);

struct SymbolTableEntry * newParameterEntry(char * id, struct SymbolTableEntry * type);
use in grammar rules

parameter_declaration:
  identifier COLON ID {
    struct SymbolTableEntry * type = findInTable(symbolTable,$1);
    struct SymbolTableEntry * entry = newParameterEntry($3,type);
    addEntryToSymbolTable(symbolTable,entry);
  } ;
use in grammar rules

identifier_list:
  ID
  initialization
  COMMA
  {
    insertLocalVariableInSymbolTable($\langle t\rangle 0, \$2, \$1);
    $\langle t\rangle $ = $\langle t\rangle 0;
  }
  identifier_list
  |  ID
  initialization
  {
    insertLocalVariableInSymbolTable($\langle t\rangle 0, \$2, \$1);
  }
  ;


void insertLocalVariableInSymbolTable(struct SymbolTableEntry * type,
    struct ConstantValue * initialization,
    char * id) {
    if (type != NULL) {
        addEntryToSymbolTable(symbolTable, newLocalEntry(id, type, initialization));
    } else {
        char * str1 = "the name ";
        char * str2 = id;
        char * str3 = " is being declared with an unknown type.";
        char * msg = (char *) malloc(strlen(str1)+strlen(str2)+strlen(str3)+1)
            * sizeof(*msg));
        msg[0] = \0;
        sprintf(msg, "%s%s%s", str1, str2, str3);
        yyerror(msg);
    }
}
Phases of a compiler

Figure 1.6, page 5 of text

Semantic analysis
Roadmap

We are going to look at examples 5.19 (p. 336) and 5.26 (p. 349) in some detail. The book revisits these examples in section 6.6.3.

Helpful background is covered in sections 5.3 and 5.4 (pages 318 through 337).
Example 5.19 (p. 335)

$S \rightarrow \text{while ( C ) } S_1$

What are the semantics of this?
Example 5.19 (p. 335)

\[ S \rightarrow \text{while ( } C \text{ ) } S_1 \]

What are the semantics of this?
Example 5.19 (p. 335)

$S \rightarrow \text{while ( C ) } S_1$

What are the semantics of this?
Example 5.19 (p. 335)

$S \rightarrow \text{while ( C ) } S_1$

What are the semantics of this?

$L_1 = \text{new()}$
$L_2 = \text{new()}$
$S_1.next = L_1$
$C.false = S.next$
$C.true = L_2$
$S.code = \text{label } || L_1 || C.code || \text{label } || L_2 || S_1.code$
Example 5.19 (p. 335)

$S \rightarrow \text{while ( } C \text{ ) } S_1$

"The inherited attribute \text{C.true} labels the beginning of the code that must be executed if \text{C} is true."

"The inherited attribute \text{S.next} labels the beginning of the code that must be executed after \text{S} is finished."

"The inherited attribute \text{S.next} labels the beginning of the code that must be executed after \text{S} is finished."

"The synthesized attribute \text{C.code} is the [code] that [implements \text{C}] and jumps either to \text{C.true} or to \text{C.false}, depending on whether \text{C} is true or false."

"The synthesized attribute \text{S.code} is the [code] that [implements \text{S}] and ends with a jump to \text{S.next}"
Figure 5.27 (p. 336)

SDD for while statement

\[
S \rightarrow \text{while ( } C \text{ ) } S_1 \{ \text{ L1 = new() } \\
L2 = \text{new() } \\
S_1.\text{next} = \text{L1 } \\
C.\text{false} = S.\text{next } \\
C.\text{true} = \text{L2 } \\
S.\text{code} = \text{label } || \text{L1 } || \text{C.code } \\
\text{|| label } || \text{L2 } || S_1.\text{code} \}
\]
Figure 5.28 (p. 336)

SDT for while statement

\[ S \rightarrow \text{while (} \{ \text{L1 = new(); L2 = new();} \]
\[ \quad \text{C.false = S.next; C.true = L2;} \]
\[ \text{)} \]
\[ \{ \text{S1.next = L1;} \]
\[ \} \]
\[ \text{S1.code = label || L1 || C.code || label || L2 || S1.code} \]
Example 5.19 (p. 335)

\[ S \rightarrow \text{while ( } C \text{ ) } S_1 \]

What are the semantics of this?

\[ S_{\text{next}} \rightarrow L1: \text{Code for } C \]
\[ L2: \text{Code for } S_1 \]
\[ C_{\text{true}} \]
\[ C_{\text{false}} \]

Diagram:
- Entry point
- Label L1:
  - Code for C
- C.true
- Label L2:
  - Code for S_1
- S.next
Example 5.26 [p. 349]

S → while ( { L1=new(); L2=new(); C.false=S.next; C.true=L2; } C ) { S1.next=L1; } S1 { S.code=label || L1 || C.code || label || L2 || S1.code}
Example 5.26 [p. 349]

S → while ( 
  M C )
N S₁ \{ S.code=label || L₁ || C.code || label || L₂ || S₁.code \} 
M → ε \{ L₁=new(); L₂=new(); C.false=S.next; C.true=L₂; \}  
N → ε \{ S₁.next=L₁; \}

? will become S on reduction

?  while  (  M
  S.next
C.true
C.false
L₁
L₂

L₁ = new(); L₂ = new();
C.true = L₂;
C.false = stack[top-3].next;
Example 5.26 [p. 349]

\[ S \rightarrow \text{while} \ (M \ C) \]
\[ N \ S_1 \{ S.\text{code}=\text{label} \ || \ L_1 \ || \ C.\text{code} \ || \ \text{label} \ || \ L_2 \ || \ S_1.\text{code} \} \]
\[ M \rightarrow \epsilon \{ L_1=\text{new}(); \ L_2=\text{new}(); \ C.\text{false}=S.\text{next}; \ C.\text{true}=L_2; \} \]
\[ N \rightarrow \epsilon \{ S_1.\text{next}=L_1; \} \]

? will become \( S \) on reduction

C can appear in many productions; \( M \) ensures that attributes are in known positions on stack
Example 5.26 [p. 349]

**Production Rules:**

- \( S \rightarrow \text{while (} \ M \ C \text{)} \ N \ S \)
- \( N \rightarrow S_1 \)
  
  \{ \text{S.code=label || L1 || C.code || label || L2 || S}_1\text{.code} \}

- \( M \rightarrow \varepsilon \)
  
  \{ \text{L1=new(); L2=new(); C.false=S.next; C.true=L2;} \}

- \( N \rightarrow \varepsilon \)
  
  \{ \text{S}_1\text{.next=L1;} \}

**Diagram:**

- '?' will become \( S \) on reduction

| \( ? \) | \text{while} | ( | \text{M} | \text{C} | ) | \text{N} |
|------|-------------|-----|-----|-----|-----|
| S.next | while | ( | M | C | ) | N |
| C.true | | | C.code | | |
| C.false | | | L1 | | |
| L2 | | | | |

\( S_1\text{.next=stack[top-3].L1} \)
Example 5.26 [p. 349]

$$S \rightarrow \text{while} ( \quad M \ C \quad ) \quad N \ S_1 \ \{ \ S\.code=\text{label} \ || \ L1 \ || \ C\.code \ || \ \text{label} \ || \ L2 \ || \ S_1\.code \}$$

$$M \rightarrow \varepsilon \ \{ \ L1=\text{new}(); \ L2=\text{new}(); \ C\.false=S\.next; \ C\.true=L2; \}$$

$$N \rightarrow \varepsilon \ \{ \ S_1\.next=L1; \}$$

? will become $S$ on reduction

<table>
<thead>
<tr>
<th></th>
<th>while</th>
<th>(</th>
<th>M</th>
<th>C</th>
<th>)</th>
<th>N</th>
<th>S_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>S.next</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C.true</td>
<td>C.code</td>
<td></td>
<td></td>
<td></td>
<td>S_1.next</td>
<td>S_1.code</td>
</tr>
<tr>
<td></td>
<td>C.false</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>L1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>L2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$S_1\.next=\text{stack}[\text{top-3}].L1$
We will revisit how the semantics of flow-of-control statements can be expressed in section 6.6.3 Flow-of-Control Statements.

At that point we will learn the backpatching approach, which you will implement in your compiler.