CSE443
Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall
Phases of a compiler

Figure 1.6, page 5 of text
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
Project Notes

How do functions work?

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

In code:

```cpp
double f(int x) { ... }
```

Diagram:

```
int x ----> f ----> double f(x)
```
Project Notes

Back to our language:

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

**function** identifier₁ `:` identifier₂ sblobk  
*Function definition.*  
identifier₁ is function name,  
identifier₂ is function type

![Diagram of function](image)
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:

description ‘;’ 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 )
Project Notes

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?
Attribute grammars

Attribute grammars provide a neater way of encoding such information.

Each syntactic rule of the grammar can be decorated with:

– a set of semantic rules/functions
– a set of semantic predicates
Attributes

- We can associate with each symbol X of the grammar a set of attributes A(X). Attributes are partitioned into:
  
  synthesized attributes S(X) – pass info up parse tree

  inherited attributes I(X) – pass info down parse tree
Example

\[ \text{\texttt{assign}} \rightarrow \text{\texttt{var}} = \text{\texttt{expr}} \]
\[ \text{\texttt{expr}}.\text{expType} \leftarrow \text{\texttt{var}}.\text{actType} \]

\[ \text{\texttt{expr}} \rightarrow \ \text{\texttt{var}}[2] + \text{\texttt{var}}[3] \]
\[ \text{\texttt{expr}}.\text{actType} \leftarrow \text{if (\text{\texttt{var}}[2].\text{actType} = \text{int}) and} \]
\[ \quad \text{(\text{\texttt{var}}[3].\text{actType} = \text{int})} \]
\[ \quad \text{then int} \]
\[ \quad \text{else real} \]
\[ \text{\texttt{expr}}.\text{actType} == \text{\texttt{expr}}.\text{expType} \]

\[ \text{\texttt{expr}} \rightarrow \ \text{\texttt{var}} \]
\[ \text{\texttt{expr}}.\text{actType} \leftarrow \text{\texttt{var}}.\text{actType} \]
\[ \text{\texttt{expr}}.\text{actType} == \text{\texttt{expr}}.\text{expType} \]

\[ \text{\texttt{var}} \rightarrow A \mid B \mid C \]
\[ \text{\texttt{var}}.\text{actType} \leftarrow \text{lookUp(\text{\texttt{var}}.\text{string})} \]

Syntactic rule

Semantic rule/function

Semantic predicate

Review
Suppose:
A is int
B is int

A = A + B

Effects of the syntactic rules is shown in red.

Everything works!
This is the same example structure, but now assume A is of type real and B is of type int.

Need for type coercion recognized during ‘+’: int → real

Generate code to do conversion.

Suppose:
A is real
B is int
Suppose: A is int
B is real

Houston, we have a problem!
Semantic predicate is **false**.

Generate error message.

Review
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]
"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.
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]
%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

e等。
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.
$n$ 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
Example: Dependency graph
(Figure 5.7)

Figure 5.7: Dependency graph for the annotated parse tree of Fig. 5.5
Semantic rules/functions

• We can associate with each rule R of the grammar a set of semantic functions.

• For rule $x_0 \rightarrow x_1 x_2 \ldots x_n$
  - synthesized attribute of LHS:
    $$S(x_0) = f(A(x_1), A(x_2), \ldots, A(x_n))$$

  - inherited attribute of RHS member:
    for $1 \leq j \leq n$, $I(x_j) = f(A(x_0), \ldots, A(x_{j-1}))$

  (note that dependence is on siblings to left only)
Reading assignment

Sections 5.3 and 5.4 (pages 318 through 337).
Example 5.19 (p. 335)

(we'll revisit in 6.6.3 on page 401)

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

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

(we'll revisit in 6.6.3 on page 401)

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 = label \| L_1 \| C.code \| label \| L_2 \| S_1.code
Example 5.19 (p. 335)

(we'll revisit in 6.6.3 on page 401)

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

There are multiple jumps out of C on both true and false to show potential short-circuiting.