CSE443
Compilers
Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall
Phases of a compiler

Semantic analysis

Figure 1.6, page 5 of text
Example 5.19 (p. 335)

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

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?
SDT for while statement

\[
S \rightarrow \text{while} ( \quad \{ \quad L1 = \text{new}(); \quad L2 = \text{new}(); \quad C.\text{false} = S.\text{next}; \quad C.\text{true} = L2; \quad \} \quad S_1.\text{next} = L1; \quad \} \quad S.\text{code} = \text{label} \ || \ L1 \ || \ C.\text{code} \ || \ \text{label} \ || \ L2 \ || \ S_1.\text{code} \quad \}
\]
Example 5.26 [p. 349]

\[
S \rightarrow \text{while} ( \{ \text{L1}=\text{new}(); \text{L2}=\text{new}(); \text{C}.false=S.\text{next}; \text{C}.true=L2; \} \\
\text{C} ) \{ \text{S}_1.\text{next}=\text{L1}; \} \\
\text{S}_1 \{ \text{S}.\text{code}=\text{label} || \text{L1} || \text{C}.\text{code} || \text{label} || \text{L2} || \text{S}_1.\text{code} \} 
\]
Example 5.26 [p. 349]

\[ S \rightarrow \text{while (} \hspace{1cm} M \hspace{1cm} C \hspace{1cm} \text{)} \hspace{1cm} N \hspace{1cm} S_1 \]  
\[ S \rightarrow S_1.\text{code} = \text{label} \parallel L_1 \parallel C.\text{code} \parallel \text{label} \parallel L_2 \parallel S_1.\text{code} \]  
\[ M \rightarrow \varepsilon \]  
\[ \{ \text{L}_1 = \text{new}(); \text{L}_2 = \text{new}(); \text{C.false} = \text{S.next}; \text{C.true} = \text{L}_2; \} \]  
\[ N \rightarrow \varepsilon \]  
\[ \{ \text{S}_1.\text{next} = \text{L}_1; \} \]  

\? will become S on reduction

\[
\begin{array}{c|c|c|c|c}
\text{stack[top-3]} & \text{stack[top-2]} & \text{stack[top-1]} & \text{stack[top]} \\
\hline
? & \text{while} & ( & M \\
S.\text{next} & & & C.\text{true} \\
& & & C.\text{false} \\
& & & L_1 \\
& & & L_2 \\
\end{array}
\]

\text{L}_1 = \text{new}(); \text{L}_2 = \text{new}();  
\text{C.true} = \text{L}_2;  
\text{C.false} = \text{stack[top-3].next;
Example 5.26 [p. 349]

\[ S \rightarrow \text{while (} \ M \ C \ \text{)} \]
\[ N \ S_1 \{ \text{S.code=label || L1 || C.code || label || L2 || S_1.code} \} \]
\[ M \rightarrow \epsilon \{ \text{L1=new(); L2=new(); C.false=S.next; C.true=L2;} \} \]
\[ N \rightarrow \epsilon \{ \text{S_1.next=L1;} \} \]

C can appear in many productions; M ensures that attributes are in known positions on stack

S.next
\[ ? \]
\[ \text{while} \]
\[ ( \]
\[ M \]
\[ \text{C} \]
\[ C.true \]
\[ C.code \]
\[ C.false \]
\[ L1 \]
\[ L2 \]

? will become S on reduction
Example 5.26 [p. 349]

\[
S \to \text{while } (M C) \\
N S_1 \quad \{ \text{S.code=\text{label} } \| \text{L1} \| \text{C.code} \| \text{label} \| \text{L2} \| \text{S}_1.\text{code} \} \\
M \to \epsilon \quad \{ \text{L1=new(); L2=new(); C.false=S.next; C.true=L2; } \} \\
N \to \epsilon \quad \{ \text{S}_1.\text{next}=\text{L1}; \} \\
\]

? will become S on reduction

\[
\begin{array}{|c|c|c|c|}
\hline
? & \text{while} & ( & M C ) \\
\hline
\hline
\text{S.next} & \text{C.true} & \text{C.code} \\
\hline
\text{C.false} & \text{L1} & \text{L2} \\
\hline
\hline
\end{array}
\]

\[
\text{stack[top-3]} \quad \text{stack[top-2]} \quad \text{stack[top-1]} \quad \text{stack[top]} \\
\]

\[
\text{S}_1.\text{next}=\text{stack[top-3].L1}
\]
Example 5.26 [p. 349]

\[
S \rightarrow \text{while (} M \ C \text{)} \\
N S_1 \{ \text{S.code=}\text{label || L1 || C.code || label || L2 || S}_1\text{.code} \} \\
M \rightarrow \epsilon \{ \text{L1=new(); L2=new(); C.false=S.next; C.true=L2; } \} \\
N \rightarrow \epsilon \{ \text{S}_1\text{.next=L1; } \}
\]

? will become $S$ on reduction
Roadmap

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.
§6.3 Types and Declarations
Type equivalence

Name equivalence: two types are equivalent if and only if they have the same name.

Structural equivalence: two types are equivalent if and only if they have the same structure. A type is structurally equivalent to itself (i.e. int is both name equivalent and structurally equivalent to int)
Name equivalence

```c
int x = 3;
int y = 5;
int z = x * y;
```

The type of `z` is `int`.
The type of `x * y` is `int`.
The names of the types are the same, so the assignment is legal.
Structural equivalence

```c
struct S { int v; double w; };
struct T { int v; double w; };

int main() {
    struct S x;
    x.v = 1; x.w = 4.5;
    struct T y;
    y = x;
    return 0;
}
```

Under name equivalence the assignment is disallowed.

Under structural equivalence the assignment is permitted.

What does C do?
C does not allow the assignment

call to the compiler:

```bash
bash-3.2$ gcc type.c
```

Output:

```bash
type.c:9:5: error: assigning to 'struct T' from incompatible type 'struct S'
    y = x;
    ^ ~
1 error generated.
```
Structural equivalence

struct S { int v; double w; };
struct T { int a; double b; };

int main() {
    struct S x;
x.v = 1; x.w = 4.5;
    struct T y;
y = x;
    return 0;
}
struct Rectangular { double x; double y; };  
struct Polar { double r; double theta; };  

int main() {  
    struct Rectangular p;  
    p.x = 3.14; p.y = 3.14;  
    struct Polar q;  
    q = p;  
    return 0;  
}
Interpretation matters

polar interpretation

rectangular interpretation
Our language
(use name equivalence)

- **built-in types:**
  - **primitive types:** integer, real, Boolean, character
  - **non-primitive type:** string

- **user-defined types:**
  - **record types have names**
    - type rec : [ real : x := 0, y := 0 ]
  - **array types have names**
    - type arr : 2 -> string
  - **function types have names**
    - type fun : ( real : x ) -> rec
Recursive records

A record type must allow a component to be of the same type as the type itself:

type Node: [ integer datum:=0 ; Node rest:=null ]
Recursive records

A record type must allow a component to be of the same type as the type itself:

```
type Node: [ integer datum:=0 ; Node rest:=null ]
```

Be careful how you process declaration: you need to ensure that the second occurrence of Node does not trigger an undefined