CSC 448: Compilers

Lecture 9
Joseph Phillips
De Paul University

2017 November 7

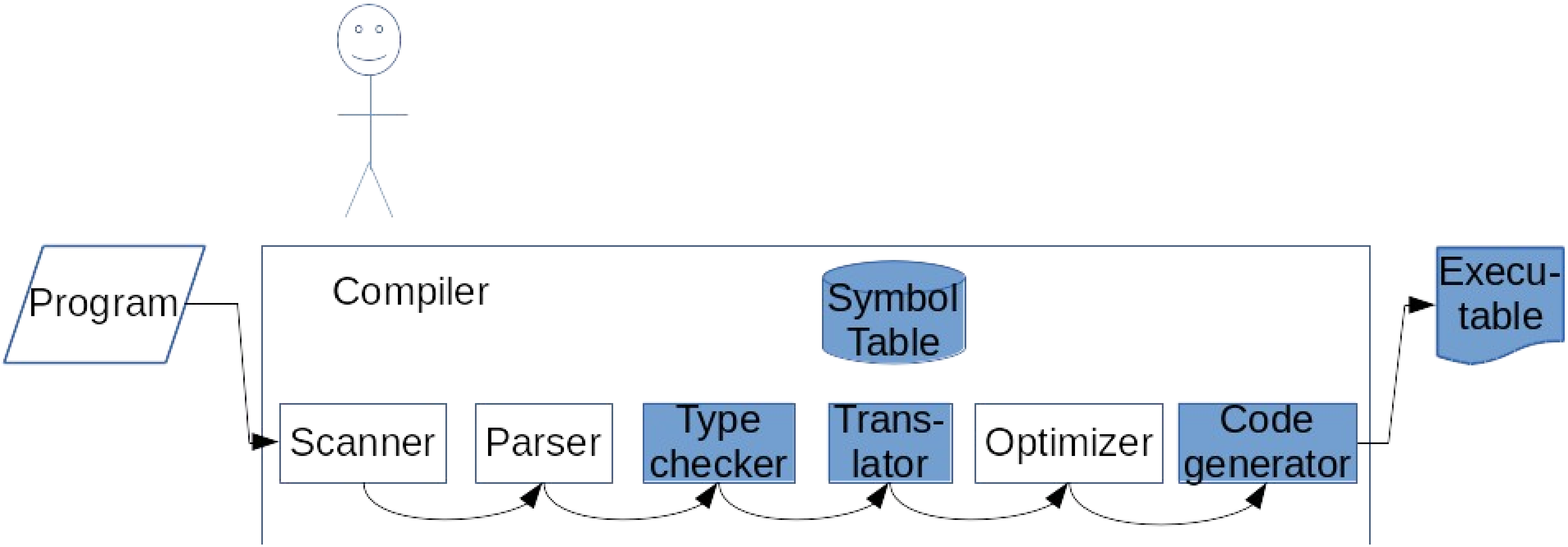
Reading

- Charles Fischer, Ron Cytron, Richard LeBlanc Jr. “Crafting a Compiler” Addison-Wesley. 2010.
  - Chapter 9: Semantic Analysis
  - Chapter 10: Intermediate Representations
  - Chapter 13: Target Code Generation

Topics:

- Semantic Analysis for Control Structures
- Intermediate Representations
- Translating Expression Trees

Overview


Program
Compiler
Symbol Table
Execu-table
Scanner
Parser
Type checker
Trans-lator
Optimizer
Code generator

Tokenized, Parsed, and Tree-ed, But where’s the executable?!?

- We still have to
  - Do semantic analysis
  -

Semantic Analysis

- What’s wrong with this code?

```
void printProcrastinator ()
{
  printf("I procrastenate printing\n");
  printf("But I’m finished now, good bye\n");
  return;
  printf("Oops!  One more thing . . .\n");
}
```

Semantic Analysis

- Or with this code?

```
void itIsImportantThatIFinish ()
{
  while (1)
    whatever();

  itIsImportantThatIBeCalled();
}
```

Two schools of thought

- If the user (the programmer) is that *stupid*, let them suffer!

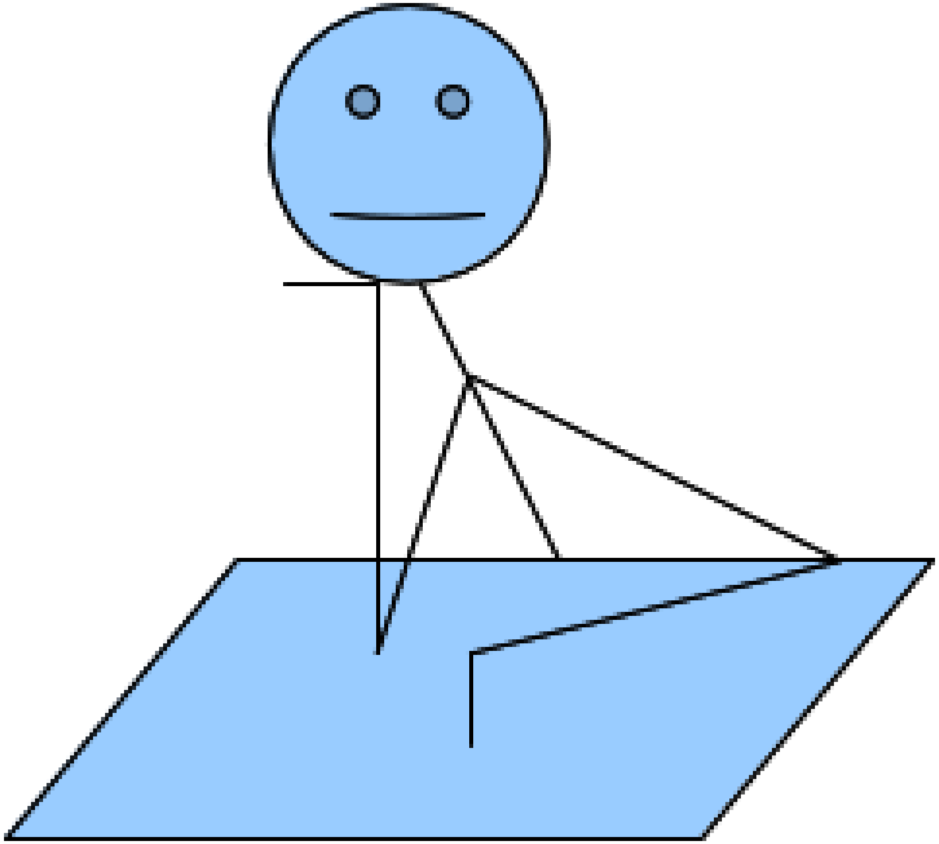
- Hey, we are the compiler!
  - Let’s try to save the user from himself/herself.

Two parse tree boolean attributes

- **`isReachable`**: Is **`true`** when code is reachable or **`false`** otherwise.
  - Compute this is a ***top-down*** fashion
- **`doesTerminateNormally`**: Is **`true`** when code can be shown to finish, or **`false`** otherwise.
  - Compute this is a ***bottom-up*** fashion
- Both are attributes of the ***parse tree nodes***.

Astute student

- “Hey! I remember from ***Computer Science Theory*** that figuring out if a program stops is ***undecidable***!”



Very true,
So we’ll be conservative about it.

We will only set the flags when we can guarantee they are true.

(In general, there will be cases we miss.)

And here are rules for computing them:

- Let’s compute **`doesTerminateNormally`** first. (Bottom-up, can do so as we parse or shortly thereafter)
- Then, we will come back and compute **`isReachable`**.

doesTerminateNormally, simple cases:

- These simple things have doesTerminateNormally value true:
  - variable declaration,
  - constant and variable evaluation
  - variable increment, decrement, simple math
  - functions already noted as terminating normally

doesTerminateNormally, statement lists:

- Assume **`doesTerminateNormally`** is **`true`** unless there is a statement for which it is **`false`**:

```
{ //  (2) so I don’t terminate normally either
  . . .
  for (;;);
  . . .
  lastThing(); // (1) Doesn’t terminate normal
}
```

doesTerminateNormally, conditionals:

- If condition and all cases terminate normally, then the conditional as a whole does too
- If the condition or one of the cases does not, then the conditional as a whole does not (unless can prove will never hit the condition)

```
if (true) // (3) Does term. normal
          // because non-normal
          // case impossible
          // to hit
{
  i++;
  // (1) simple op, so
  // does terminate normal
}
else
{
  while (true);
  //  (2) Does NOT terminate
  //  normally, but will
  //  never get here
}
```

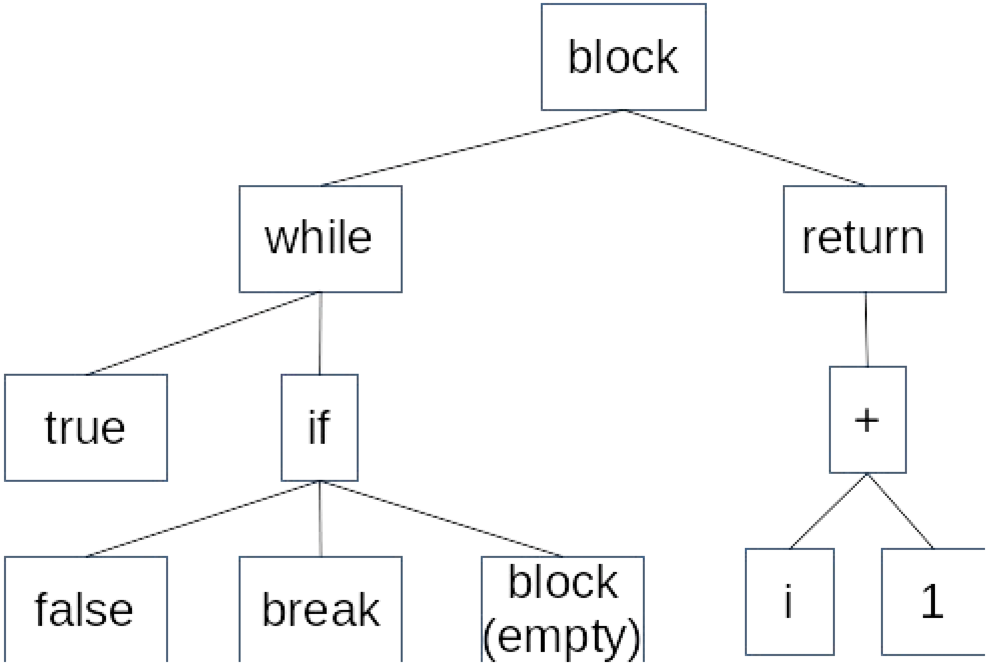
doesTerminateNormally, loops:

- Is **`false`** if looks like infinite loop:
  - `while (true) ...`
  - `for (;;) ...`
- However, look inside body. Could be **`true`** if there is a **`break`** statement that you could hit.

```
while (true)
// (1) Looks like infinite loop
// (4) So true is infinite loop
// => doesTerminateNormal=false
{
  if  (false)
      // (3) But will never hit
  {
    break;
    // (2) But there is a break
  }
}
```

Example
(1) Create parse tree

```
int screwy (int i)
{
  while (true)
  {
    if  (false)
      break;
  }
  return(i + 1);
}
```



block

while

return

true

if

+

false

break

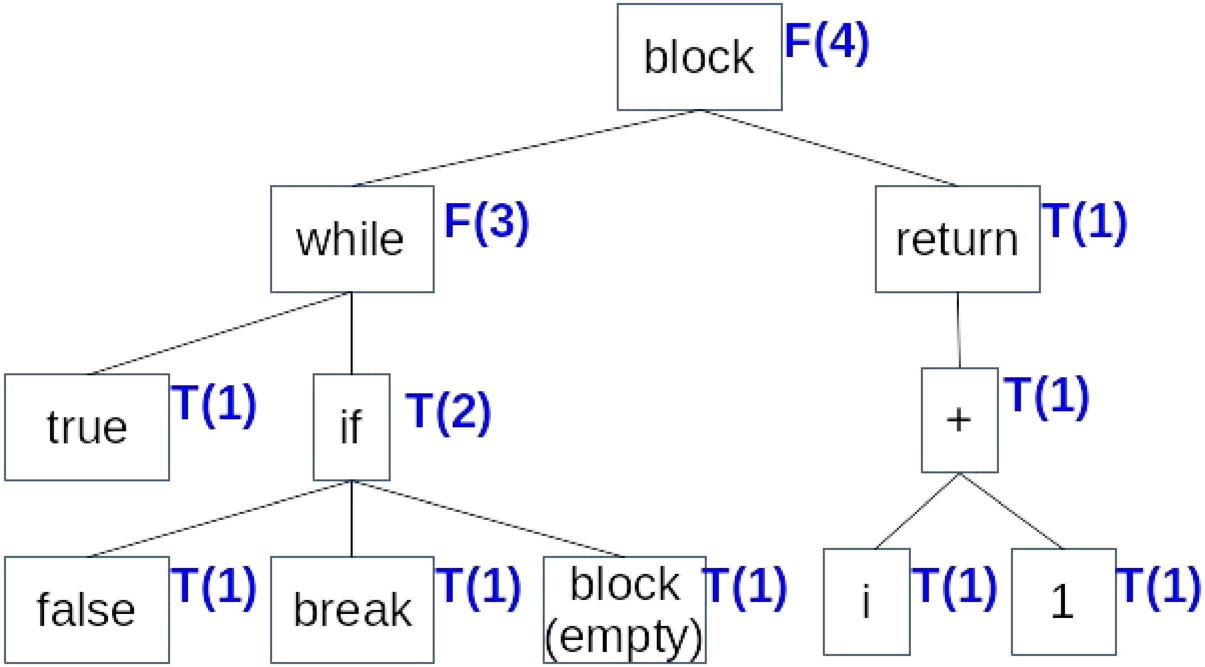
block
(empty)

i

1

Example (2) Compute doesTerminateNormally

```
int screwy (int i)
{
  while (true)
  {
    if  (false)
      break;
  }
  return(i + 1);
}
```



(1) Simple cases
(2) By conditional rule
(3) By loop rule
(4) By statement list rule

doesTerminateNormally in {T,F}

Now compute isReachable in top-down fashion

isReachable, statement lists (1):

- If **`isReachable`** is **`true`** for a statement list, then it is also **`true`** for the first statement in the list

```
{                // If I am reachable
  firstThing(); // then I am reachable too
  . . .
}

//  Functions are a special case of this
void    foo  ()
{                // I start out being reachable
  firstThing();
  . . .
}
```

isReachable, statement lists (2):

- A statement in a statement list has **`isReachable`** value equal to the **`doesTerminateNormally`** value of the statement before it in the list

```
{
  . . .
  for (;;); // I don’t terminate normally
  nextThing(); // therefore, I’m not reachable
  . . .
}
```

isReachable, conditionals:

- If condition excludes a case then have unreachable code

```
if (true)
{
  i++;
  // (1)isReachable= true
}
else
{
  while (true);
  // (2)isReachable=false
}
```

isReachable, loops:

- Look for loops that are never taken:
  - `while (false) {}`
  - `for (;false;) {}`

```
while (false)
{
  // Nothing in
  // here gets
  // executed
}
```

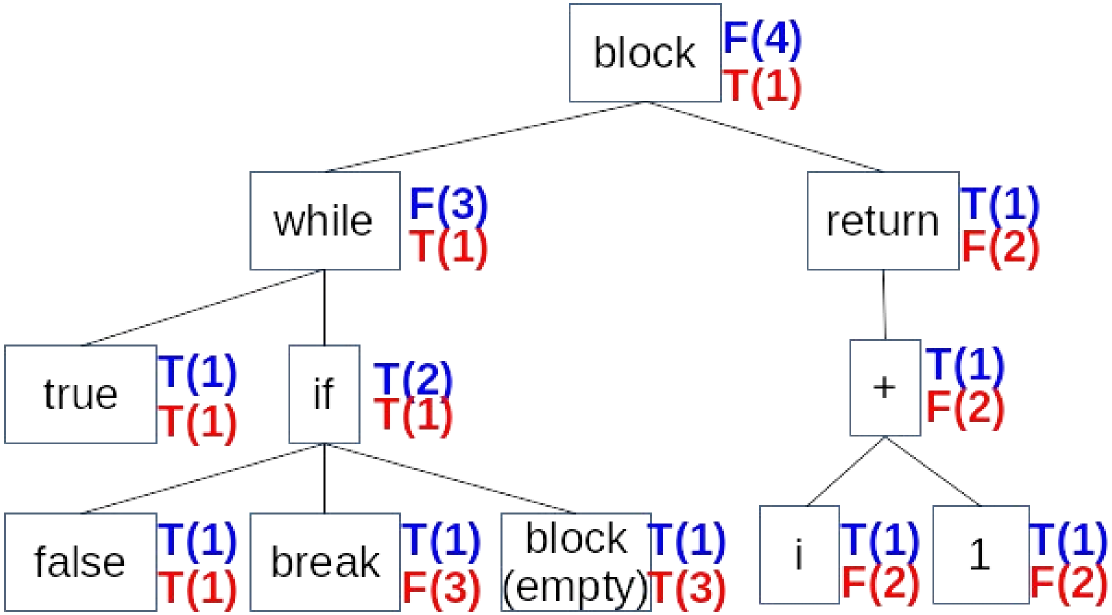
isReachable, shall we complain?

- It is not an error if an empty statement has **`isReachable`** value **`false`**

```
void    foo  ()
{
  while (false)
  {
    //  I’m never done, but I don’t do anything anyway
  }
  whatever();
  return;
  ;  // I’m not reachable, but no worries
  {} // Same for me.
}
```

Example (2) Compute isReachable

```
int screwy (int i)
{
  while (true)
  {
    if  (false)
      break;
  }
  return(i + 1);
}
```



(1) By statement list-1
(2) By statement list-2
(3) By conditional rule
(4) By loop rule

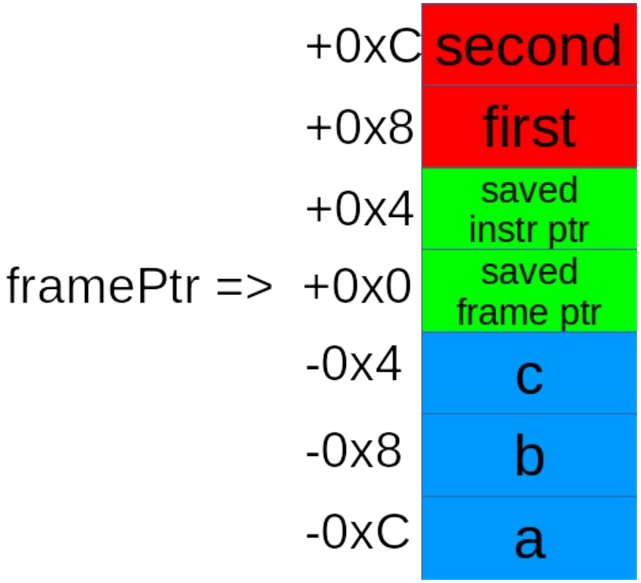
doesTerminateNormally in {T,F}
isReachable in {T,F}

Semantic analysis of function/method calls

- Account for:
  - Function/method name
  - Inheritance from superclasses
  - **`private`** or **`protected`** context
  - Number and type of arguments
  - Return type
  - (In C++) **`const`**-ness of method

Generating assembly language
The C stack frame

```c
int foo(int first,
        int second
       )
{
   int a = 0;
   int b = 1;
   int c = 2;
   return(10);
}
```



	+0xC	second
	+0x8	first
	+0x4	saved instr ptr
framePtr =>	+0x0	saved frame ptr
	-0x4	c
	-0x8	b
	-0xC	a

```
push    %ebp
mov     %esp,%ebp
sub     $0x10,%esp
movl    $0x0,-0xc(%ebp)
movl    $0x1,-0x8(%ebp)
movl    $0x2,-0x4(%ebp)
mov     $0xa,%eax
leave
ret
```

Function and method calls

- Either caller() or callee() must save registers on stack
  - Compiler can keep track of which registers the callee() uses (only they need be saved)
  - global vars in registers should be written back to memory (so callee() sees most recent value)

- Also, functions may have a prologue:

```
push    %ebp
mov     %esp,%ebp
sub     $0x10,%esp
```

- And an epilogue:

```
leave
ret
```

Register allocation