This Lecture

- Limitations of context-free languages: Why checking contextual constraints is different from checking syntactical constraints.
- Identification (or Name Resolution)
- Block Structure
- Symbol table

Contextual Analysis (1)

Our next major topic is contextual analysis or checking static semantics.

Among other things, this involves:
- Resolve the meaning of symbols.
- Report undefined symbols.
- Type checking.

Contextual Analysis (2)

In short, contextual analysis is about ensuring that a program is statically well-formed.

- But syntax has to do with “form” too. So what is new?
- Can’t we use context-free grammars (CFG) to express e.g. type constraints and thus make the parser do the checking for us?

E.g., grammar productions like:

```
Cmd → if BoolExpr then Cmd
```

Limitations of CFGs (1)

Attempt to express a “declare before use” requirement using a CFG. Assumption: only a single variable a:

```
Prog → DecA ProgA | DecB ProgB
ProgA → StmtA ProgA | DecAB ProgAB | e
ProgB → StmtB ProgB | DecAB ProgAB | e
ProgAB → StmtAB ProgAB | e
DecA → int a;
DecB → int b;
StmtA → a = ExprA;
StmtB → b = ExprB;
StmtAB → (a | b) = ExprAB;
ExprA → a | ExprA + ExprA | Expr
ExprB → b | ExprB + ExprB | Expr
ExprAB → a | b | ExprAB + ExprAB | Expr
Expr → LitInt | Expr + Expr
```

Limitations of CFGs (2)

Generalization to two variables, a and b:

```
Prog → DecA ProgA | DecB ProgB
ProgA → StmtA ProgA | DecAB ProgAB | e
ProgB → StmtB ProgB | DecAB ProgAB | e
ProgAB → StmtAB ProgAB | e
DecA → int a;
DecB → int b;
StmtA → a = ExprA;
StmtB → b = ExprB;
StmtAB → (a | b) = ExprAB;
ExprA → a | ExprA + ExprA | Expr
ExprB → b | ExprB + ExprB | Expr
ExprAB → a | b | ExprAB + ExprAB | Expr
Expr → LitInt | Expr + Expr
```

Limitations of CFGs (3)

Some observations:

- Already for two variables, things get quite complicated.
- In fact, the number of nonterminals grow exponentially. E.g., for a set of n variables \( V = \{ a_i | 1 \leq i \leq n \} \), we get \( 2^n \) nonterminals \( Expr[W] \), one for each \( W \subseteq V \).
- Normally, the number of variables is unlimited. That would imply infinitely many productions. No longer a CFG!

Limitations of CFGs (4)

Attempt to describe simple type constraints using a CFG:

```
IntExpr → LitInt | IntVar
BoolExpr → false | true | BoolVar | IntExpr + IntExpr
```

Where Are We?

1. Sequence of characters
2. Lexical Analysis
3. Syntactic Analysis/Parsing
4. Contextual Analysis/Checking Static Semantics (e.g. Type Checking)
5. Optimization and Code Generation (possibly many steps involving a number of intermediary representations)
6. Target code
Limitations of CFGs (5)

Might look reasonable at first sight. However:

- The scheme hinges on partitioning the variables by name into two groups: integer variables (IntVar) and boolean variables (BoolVar).
- But in most languages the type of a variable is given by the context, not its name.
- And how could we in general infer argument types from the name of a procedure or function?
- We should not expect to be able to capture context-sensitive information using a context-free grammar.

Unrestricted Grammars (1)

- These examples do not prove that it is impossible to achieve what we tried to achieve using CFGs.
- However, it can be proved that this indeed is the case: contextual constraints result in context-sensitive or even recursively enumerable languages; such languages cannot be described by CFGs.

Contextual Analysis (1)

Two important kinds of contextual constraints:

- **Scope rules**: visibility; which declarations take effect where.
- **Type rules**: internal consistency; ensuring that every expression computes a value of acceptable form, i.e., has a valid type.

These are the ones we mainly will be concerned with in this course.

Unrestricted Grammars (2)

- **Unrestricted grammars** with productions $\alpha \rightarrow \beta$
  
  where $\alpha$ and $\beta$ both are arbitrary strings could be used to express arbitrary contextual constraints.
- However, unrestricted grammars are in fact equivalent to Turing Machines!
- Neither Turing Machines nor Unrestricted Grammars are very practical languages: a common choice is a general-purpose language, like C, Java or Haskell.

Contextual Analysis (2)

Corresponding subphases of the contextual analysis:

- **Identification or Name Resolution**: applying the scope rules in order to relate each applied identifier occurrence to its declaration.
- **Type checking**: applying the type rules to infer the type of each expression, and compare it with the expected type.

Contextual Analysis (3)

Many other possible kinds of contextual constraints. E.g. Java has rules concerning:

- **Abstract classes**: e.g.:
  - Only abstract classes can have abstr. methods:
    ```java
    abstract class A {
      abstract void callme();
    }
    ```
  - Abstract classes may not be instantiated:
    ```java
    Not allowed: new A();
    ```

Contextual Analysis (4)

- **Final classes**: e.g.:
  - A final class cannot be extended
  - A class cannot be both final and abstract.
- **Exceptions**: e.g., the set of exceptions a method can raise must be declared (except for unchecked exceptions):
  ```java
  public void writeList() throws IOException { ... }
  ```

Contextual Analysis (5)

- **Definite assignment**: a local variable must not be read unless it has been “definitely assigned before”.
  For example, the code fragment
  ```java
  int k, n = 5;
  if (n > 2) k = 3;
  System.out.println(k);
  ```
  is rejected.

An example of a Java "definite assignment" rule:

$V$ is definitely assigned after

```java
if (r) S else T
```

Note: The rule does not take the ultimate run-time value of $r$ into account. That is what makes the analysis decidable.
Identification (or Name Resolution) is the task of relating each applied identifier occurrence to its declaration.

```java
public class C {
    int x, n;
    void set(int n) { x = n; }
}
```

In the body of `set`, the one applied occurrence of
- `x` refers to the instance variable `x`
- `n` refers to the argument `n`.

### Scope and Scope Rules (1)

The identification process is governed by the scope rules of the language.

Important terms:
- **Scope**: the portion of a program over which a declaration takes effect.
- **Block**: a program phrase that delimits the scope of declarations within it.

### Scope and Scope Rules (2)

Consider the MiniTriangle `let` block command:
```java
let decls in body
```

The scope of each declaration is the rest of the block.

For example:
```java
let
    const m = 10;
    const n = m * 2
in
    putint(n);
```

### Scope and Scope Rules (3)

Haskell's `let`-expressions:
```haskell
let id = expr in body
```

The scope of `id` includes both `expr` and `body`.

For example:
```
let xs = 1:xs in take 7 xs
```

### Scope and Scope Rules (4)

Part II of the coursework uses a version of Mini-Triangle extended with procedures and functions:
```java
let
    const n : Integer = 2;
    proc p(x : Integer) begin
        ...
        p(x * m) ...
    end;
    const m : Integer = n * n
in
    ...
```

### Scope and Scope Rules (5)

In the extended version:
- The scope of a declared entity is extended to include the bodies of all procedures and functions declared in the same `let`-block.
- This allows procedures and functions to be (mutually) recursive.
- However, definition/initialization expressions for constants/variables must not use functions defined in the same `let`-block.
- This avoids calling functions that may refer to as-yet uninitialized variables.

### Scope and Scope Rules (6)

In addition to deciding the range of declarations, the scope rules also also deal with issues like
- whether explicit declarations are required
- whether multiple declarations at the same level are allowed
- whether shadowing/hiding is allowed.

### Some Java Scope Rules (1)

From the Java Language Specification ver. 1.0:
- The scope of a member declared in or inherited by a class type or interface type is the entire declaration of the class or interface type. The declaration of a member needs to appear before it is used only when the use is in a field initialization expression.
- The scope of a parameter of a method is the entire body of the method.

```java
static int x = 10;
public void foo(int x) {
    if (x < 0) {
        int x = 10;
        ...
    }
    ...
}
```

- Hiding the name of a local variable is not permitted. For example, the following code fragment is rejected:
  ```java
  static int x = 10;
  public void foo(int x) {
      if (x < 0) {
          int x = 10;
          ...
      }
      ...
  }
  ```
A **symbol table**, also called **identification table** or **environment**, is used during identification to keep track of **symbols** and their **attributes**, such as:
- kind of symbol (class name, local variable, etc.)
- scope level
- type
- source code position

**Using the Symbol Table**
For a simple language with a declare-before-use rule, redeclarations not allowed, the symbol table would be used as follows during identification:
- Initialise the table; e.g., enter the standard environment.
- When a declaration is encountered:
  - check if declared identifier clashes with existing symbol
  - report error if it does
  - if not, enter declared identifier into table along with its attributes.

**Using the Symbol Table**
After identification:
```plaintext
let int x = 1
in
let int y = x * 3
in
x + y
```
(The textual representation of annotated AST.)

**Using the Symbol Table**
Suppose variables have to be declared, and that redeclarations are not allowed.
```plaintext
let int x = 1; int x = x * 3
in
x + y
```
During symbol table insert and lookup it would be discovered that:
- `x` is declared twice at the same scope level,
- `y` is not declared at all.

**Using the Symbol Table**
Suppose variables have to be declared, and that redeclarations are not allowed.
```plaintext
let int x = 1; int x = x * 3
in
x + y
```
During symbol table insert and lookup it would be discovered that:
- `x` is declared twice at the same scope level,
- `y` is not declared at all.

**Using the Symbol Table**
Before identification:
```plaintext
let int x = 1
in
let int y = x * 3
in
x + y
```

**Using the Symbol Table**
We will focus on nested block structure in the following since:
- monolithic and flat block structure can be considered special cases of nested block structure
- variations on nested block structure is by far the most common in modern high-level languages.

**Using the Symbol Table**
Before identification:
```plaintext
1 let int x = 1
2 in
3 let int y = x + 3
4 in
5 x + y
```

**Using the Symbol Table**
When entering a new block, arrange so that subsequently entered symbols become associated with the scope corresponding to the block (“open scope”).
- When leaving a block, remove/make inaccessible symbols declared in that block (“close scope”).
Using the Symbol Table (7)

```haskell
let int x = 1
in (let y = x * 3 in x) + y
```

- A new scope is opened for the inner let-block when it is analysed.
- When the inner let-block has been analysed, its scope is closed.
- It is then discovered that y is no longer in scope. (However, x is still in scope.)

Summary

- Contextual analysis includes checking scope rules and types.
- Contextual constraints lead to context-sensitive languages and thus cannot be captured by a context-free grammar.
- **Identification** is the task of relating each applied identifier occurrence to its declaration. A key step for any contextual analysis.
- The **Symbol Table** or **Environment** records information about declared entities and is the central data structure during contextual analysis.