Cecko

Assignment 3 to 5 – Semantic analysis

Base and bonus assignments

Assignment 3 – declarations	Assignment 4	Assignment 5
100: Types: Built-in, TYPEIDF, const Pointer, Function Declarations: Variable, Function Expressions: Integer literals Statements: Return	100: Conversions: Array-to-pointer _Bool/char-to-int, int-to-char Expressions: char/string literals, Variables Function call (also variadic) Unary +, -, *, & Binary int +,-,*,/,% Binary char/int/ptr = Statements: expression, return	100: Conversions: any-to-_Bool Expressions: int <,<=,>,>=,==,!= Unary ! _Bool = _Bool arguments and return values Statements: if,while,do
+10: Declarations: typedef	+10: sizeof	+10: Statements: for
+10: Types: Array	+10: Expressions: int ++, --, +=, -=, *=, /=, %= ptr ++, --, +=, -=	+10: Expressions: ptr <,<=,>,>=,==,!= ptr-ptr
+20: Types: Enum	+20: Expressions: ptr+int, int+ptr, ptr-int ptr[int], int[ptr]	+30: Expressions: Binary ||,&&
+20: Types: Struct	+10: Expressions: .,->, struct = Struct arguments and return values	+6: Bonus for passing extra tests

Semantic actions in bison

bison

- a bison parser can evaluate a **purely-synthesized attribute grammar**
  - **synthesized attributes** are passed from children to parents
    - i.e. from the right-hand-side of a rule to the nonterminal on the left
  - inherited attributes cannot be supported by bison
    - LALR(1) parser builds the tree bottom-up, during reductions
  - bison supports only one attribute for each non-terminal
    - it may be a C++ structure
    - each non-terminal may produce a different type of attribute
- attribute types are declared in the first section of a grammar file
    - declaration of the terminal A and its attribute type T

```
%token<T> A
```

    - declaration of the attribute type T of the non-terminal a

```
%type<T> a
```

- attribute values are assigned in the C++ code for each rule
    - example: evaluating an expression (in compile time)

```
%type<double> mul_expr add_expr
%%
add_expr:
   mul_expr { $$ = $1; }
   | add_expr ADD mul_expr { $$ = $1 + $3; }
   | add_expr SUB mul_expr { $$ = $1 - $3; }
   ;
```

- Technical details

```
%token<T> A
%type<T> a
```

  - The T must be a class/type (qualified) identifier in C++
    - container instances, pointers, etc. must be named by typedef/using
  - The T must support default constructor and copy-assignment
    - data containing unique_ptr cannot be used
  - In the case of syntax-error recovery, attributes are discarded
    - instead of being used in a rule
  - $$ denotes the attribute of the LHS non-terminal
    - the output attribute of the rule
    - it shall be assigned by the C++ code of the rule
      - it may be assigned by parts ($$.a) or accessed repeatedly in the code fragment
  - $i denotes the attribute of the i-th symbol on the RHS
    - every symbol counts, even if it has no attribute
  - @i denotes the location (line number) of the i-th symbol on the RHS
    - for non-terminals, it is automatically computed from terminals contained
  - if there is no C++ code in a rule, `{ $$ = $1; }` is used automatically
    - it may fail due to type incompatibility
    - not a good practice to rely on this

- a bison parser can evaluate a **purely-synthesized attribute grammar**
  - it is sufficient for compiling expressions and declarations
- it cannot pass the information from the declaration to a use of an identifier
  - instead of bison attributes, global state (ctx) must be used
  - we must understand the order in which the C++ code fragments are executed
    - a post-order pass through the (physically non-existent) derivation tree
  - special care needed in some cases
    - example: linked list declaration

```
struct N { int v; struct N * next; };
```

      - the declaration of struct N must appear in symbol tables before entering the { }
      - the full definition of struct N must be added to symbol tables afterwards

```
decl_head:
   STRUCT IDF { $$ = ctx->struct_definition_open($2); }
decl:
   decl_head LCUR elem_list RCUR { ctx->struct_definition_close($1,$3); }
   ;
```

    - note: bison can somehow handle code fragments inside the RHS
      - their presence has surprising side-effects
      - rewriting grammar to keep code at the end of RHS is preferred
  - in Assignments 4 and 5, we will need similar tricks to handle code generation

bison & flex

- bison is a LALR(1) parser – look-ahead of 1 token
  - when parser does a reduce, lexer has already processed the next token
- the TYPEIDF trick in the lexer

```
[a-z]+    { if ( ctx->is_typedef(yytext) )
                return make_TYPEIDF( yytext);
             else return make_IDF( yytext); }
```

  - Example:

```
typedef int * ptr ; ptr x ;
```

  - In the parser, this rule...

```
declaration: TYPEDEF type declarator SEMIC { ctx->define_typedef($2, $3); };
```

  - ... would NOT work
    - the second "ptr" is already returned from the lexer as IDF when define_typedef is called
  - The correct solution is:

```
declaration_core:
   TYPEDEF type declarator { ctx->define_typedef($2, $3); };
declaration: declaration_core SEMIC;
```

  - The real grammar allows a sequence of declarators – it may be rewritten as:

```
declaration_core:
   TYPEDEF type declarator { ctx->define_typedef($2, $3); $$ = $2; }
   | declaration_core COMMA declarator { ctx->define_typedef($1, $2); $$ = $1; }
   ;
```

    - in reality, it must allow more variants than "TYPEDEF type" including "int typedef const"
  - Another example – scope exit must be reported *before* the closing parenthesis

```
void f(); int x; void g() { { typedef char f; } f(x); }
```

A minimal introduction to LLVM IR

LLVM IR

- LLVM Intermediate Representation
  - The output of your work (directly or indirectly through the cecko framework)
    - Global variables (including string constants)
    - Functions containing code (including instructions allocating local variables)
- llvm::Value – Abstract class representing any "algorithm" providing a value
    - Held by llvm::Value * = cecko::CKIRValueObs
  - llvm::Constant
    - llvm::ConstantInt – integer constants computed during compilation
      - Held by llvm::ConstantInt * = cecko::CKIRConstantIntObs
    - other immutable values like addresses of global variables
  - llvm::Instruction – anything computed at run-time
    - including llvm::AllocaInst - allocate an address for a local variable
      - Automatically generated by the cecko framework
    - created using llvm::IRBuilder - provides automatic constant folding
      - if an instruction being created has constant operands, the result constant is created instead
- llvm::Type – Abstract class representing a type of a value
    - Held by llvm::Type * = cecko::CKIRTypeObs
  - Simplified type system of the C language
    - arbitrarily sized integers, floats, pointer, arrays, structs, functions
    - no enums, no signed/unsigned flag, no const flag – insufficient for compiling C

The cecko framework for Assignment 3

cecko framework

- The cecko framework provides
  - A type system corresponding to the cecko language
    - void, _Bool, char, int – the built-in types
    - pointer, array, function
    - struct, enum
    - const flags where required
  - Named entities in two name spaces
    - Identifiers. Due to the TYPEIDF trick, divided into two subspaces
      - Named objects – enum constants, variables, functions – IDF tokens
      - Typedefs – defined from IDF tokens, used as TYPEIDF tokens
    - Tags – structs and enums
      - You shall allow both IDF and TYPEIDF after struct/enum keywords!
  - Both name spaces are divided into scopes (i.e. compound statements)
    - The framework automatically handles scoping rules
      - Your responsibility is to report scope boundaries
    - Scopes also provide automatic distinction between global and local variables
      - Function arguments are automatically converted into local variables
  - structs, enums and functions have distinct declarations and definitions
  - Where applicable, the framework also creates the corresponding LLVM entities

cecko framework

- All creation/modification services of the framework provided through ctx
  - hold by cecko::context_obs
  - member functions of cecko::CKContext
  - member functions of cecko::context (messaging functions)
- Inspecting entity properties is done via the corresponding (abstract) class
  - cecko::CKAbstractType – all type descriptors
    - hold by cecko::CKTypeObs
  - cecko::CKAbstractNamed – constants, variables, functions
    - hold by cecko::CKNamedObs
  - cecko::CKTypedef – typedefs
    - hold by cecko::CKTypedefConstObs
- For many entities, there are two types of pointers, e.g.:
  - cecko::CKTypeObs = const cecko::CIAbstractType *
    - Dereference (->) will crash when null
  - cecko::CKTypeSafeObs
    - Dereference (->) is safe (provides a dummy object) when null
    - Use ! to test for null
    - This allows easy recovery from many semantic errors
    - Implicitly convertible to cecko::CKTypeObs

Examples

- typedef const int * strange[3];
  - int

```
auto t1=ctx->get_int_type();
```

  - const

```
auto tp1=CKTypeRefSafePack(t1,true);
```

  - 3

```
auto value=ctx->get_int32_constant(3);
```

  - *

```
auto t2=ctx->get_pointer_type(tp1);
auto tp2=CKTypeRefSafePack(t2,false);
```

  - []

```
auto t3=ctx->get_array_type(tp2.type,value);
auto tp3=CKTypeRefSafePack(t3,tp3.is_const);
```

  - strange

```
ctx->define_typedef(“strange”,tp3,loc);
```