TÉCNICO LISBOA

Compiladores[1]

Análise Semântica

Capítulo 7 - "Compiladores: Da Teoria à Prática"

Prof. Alberto Abad

IST - Universidade de Lisboa

2021/2022

[1] Slides adaptados de Prof. Pedro T. Monteiro (2017/2018)

Identificador indevido

```
#include <iostream>

int main() {
        std::cout << "Exemplo erro lexical:" << std::endl;
        int a?bc = 3;
        return 0;
}
```

- Erro(s) de compilação?

TÉCNICO LISBOA

Utilização indevida de palavras chave

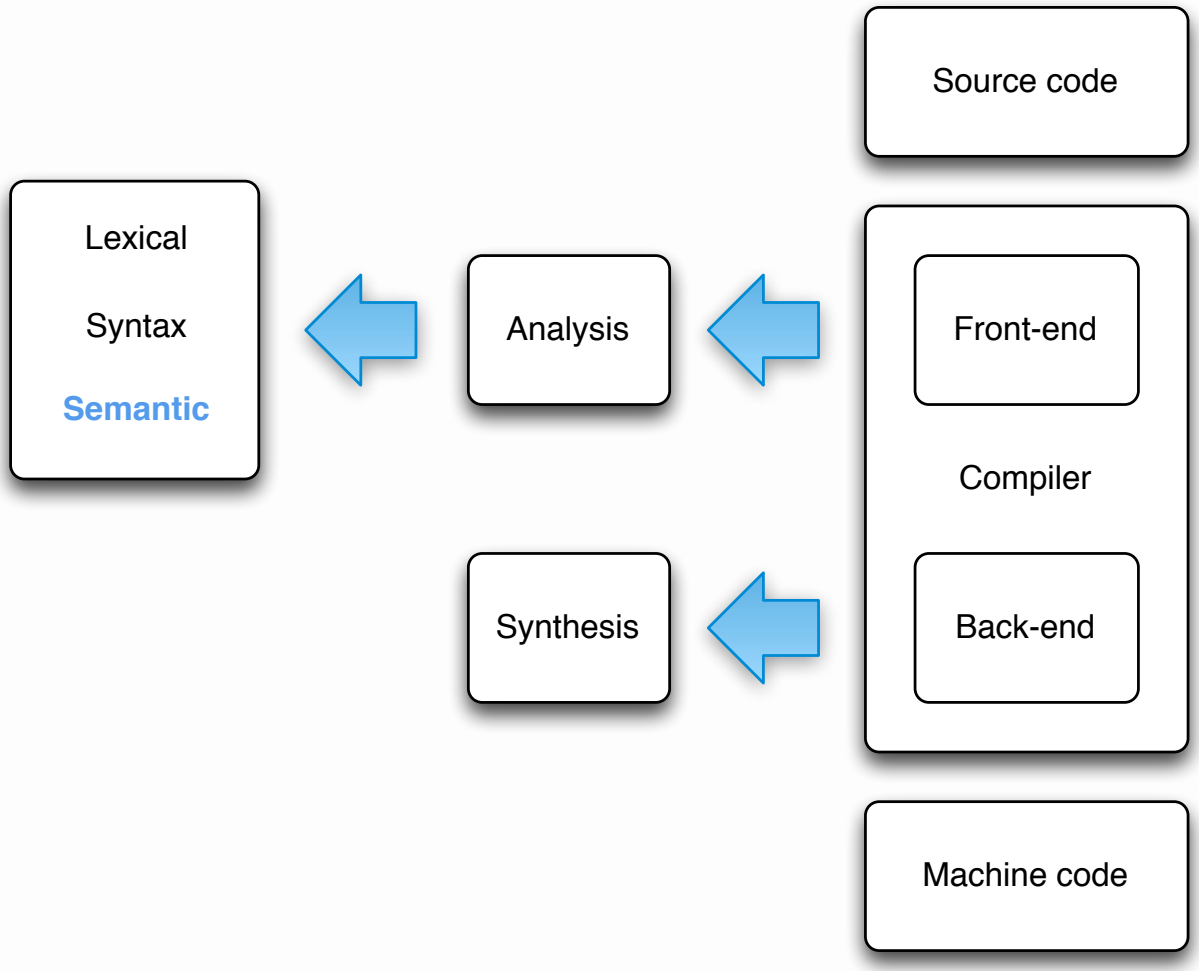
```
#include <iostream>

int main() {
        std::cout << "Exemplo erro sintatico:" << std::endl;
        int a = while + 5;
        return 0;
}
```

- Erro(s) de compilação?

Outline

Análise semântica

Fases de um compilador



Análise semântica

Depois da análise léxica e sintáctica, um programa pode ainda conter erros ao nível semântico

Análise semântica

Depois da análise léxica e sintáctica, um programa pode ainda conter erros ao nível semântico

Problemas:

- Verificações de tipos: expressões, funções, etc.
- Declarações de identificadores: variáveis, funções, classes, métodos, etc.
- Verificações de contexto: `continue`, `break`, etc.
- Outros: verificações de tags, fluxos de controlo, *lvalues* em atribuições, acesso a membros de uma classe, etc.
- ...

Análise semântica

Problemas

```
#include <iostream>

int f() {
        std::cout << "Exemplo 1:" << std::endl;
        int a = "hello world" + 5;
        return 0;
}

int g() {
        std::cout << "Exemplo 2: " << -"really!?" << std::endl;
        return 0;
}
```

- Erro(s) de compilação?

Verificações de tipos em expressões

```
#include <iostream>

int f() {
        std::cout << "Exemplo 1:" << std::endl;
        int a = "hello world" + 5;
        return 0;
}

int g() {
        std::cout << "Exemplo 2: " << -"really!?" << std::endl;
        return 0;
}
```

- Erro(s) de compilação?

Análise semântica

Problemas

```
#include <iostream>

void f(int a) {
        std::cout << a << std::endl;
}

int main() {
        std::cout <<  "Exemplo 3:" << std::endl;
        f("hello world");
        return 0;
}
```

- Erro(s) de compilação?

Verificações de tipos (argumentos) em chamadas a funções

```
#include <iostream>

void f(int a) {
        std::cout << a << std::endl;
}

int main() {
        std::cout << "Exemplo 3:" << std::endl;
        f("hello world");
        return 0;
}
```

- Erro(s) de compilação?

Análise semântica

Problemas

```
#include <iostream>

int main() {
        std::cout <<  "Exemplo 4:" << std::endl;
        int a =  3;

        if (a > 0)
                int b = 2;

        std::cout << a + b << std::endl;
        return 0;
}
```

- Erro(s) de compilação?

Declaração identificadores: variáveis

```
#include <iostream>

int main() {
        std::cout <<  "Exemplo 4:" << std::endl;
        int a =  3;

        if (a > 0)
                int b = 2;

        std::cout << a + b << std::endl;
        return 0;
}
```

- Erro(s) de compilação?

Análise semântica

Problemas

```
#include <iostream>

int main() {
        std::cout << "Exemplo 5:" << std::endl;

        return f();
}
```

- Erro(s) de compilação?

Declaração identificadores: funções

```
#include <iostream>

int main() {
        std::cout << "Exemplo 5:" << std::endl;

        return f();
}
```

- Erro(s) de compilação?

Análise semântica

Problemas

```
#include <iostream>

void f() {
        std::cout << "do Something" << std::endl;
}

void f() {
        std::cout << "do Something different?" << std::endl;
}

int main() {
        std::cout <<  "Exemplo 6:" << std::endl;
        f();
        return 0;
}
```

- Erro(s) de compilação?

Declaração identificadores: declarações múltiplas

```
#include <iostream>

void f() {
        std::cout << "do Something" << std::endl;
}

void f() {
        std::cout << "do Something different?" << std::endl;
}

int main() {
        std::cout <<  "Exemplo 6:" << std::endl;
        f();
        return 0;
}
```

- Erro(s) de compilação?

Análise semântica

Problemas

```
#include <iostream>

int main() {
        std::cout << "Exemplo 7:" << std::endl;

        int i = 0;
        if (i < 10)
                break;

        return 0;
}
```

- Erro(s) de compilação?

Instruções em contexto errado

```
#include <iostream>

int main() {
        std::cout << "Exemplo 7:" << std::endl;

        int i = 0;
        if (i < 10)
                break;

        return 0;
}
```

- Erro(s) de compilação?

TÉCNICO LISBOA

Como implementar?

Análise semântica

Como implementar?

- Tabela(s) de símbolos
  - Permite representar identificadores
  - Permite resolver questões relativas a scopes
- Árvores de sintaxe abstracta (abstract syntax trees, ASTs)
  - Type checking basedo no padrão de desenho Visitor
  - Possibilidade de realizar vários passos sobre o código
  - Possibilidade de ter controlo do contexto

Outline

Símbolos

- Classe do identificador:
  - Variável, função, etc.
- Tipo do identificador
  - int, bool, real, ...
- Outras características:
  - Constante
  - Global ou local
  - Argumentos em funções
  - Linkage (internal, external)
  - ...

Símbolos

Relação com o projecto

l22/targets/symbol.h

```
#include <string>
#include <memory>
#include <cdk/types/basic_type.h>

namespace l22 {

  class symbol {
    std::shared_ptr<cdk::basic_type> _type;
    std::string _name;
    long _value; // hack!

  public:
    symbol(std::shared_ptr<cdk::basic_type> type, const std::string &name, long value) :
        _type(type), _name(name), _value(value) {
    }

    virtual ~symbol() {  /* EMPTY */  }

    std::shared_ptr<cdk::basic_type> type() const { return _type;}
    bool is_typed(cdk::typename_type name) const { return _type->name() == name;}
    const std::string &name() const { return _name;}
    long value() const { return _value;}
    long value(long v) { return _value = v;}
  };

}
```

Outline

Tabela de símbolos

Objectivo:
Representação e gestão dos símbolos de um programa

Interface básica:

- insert(symb): inserir símbolo na tabela
- find(symb): procurar símbolo na tabela

Tabela de símbolos

Relação com o projecto

/usr/include/cdk/symbol_table.h

Insert

```
bool insert(const std::string &name, std::shared_ptr<Symbol> symbol) {
  auto it = _current->find(name);
  if (it == _current->end()) {
    (*_current)[name] = symbol;
    return true;
  }
  return false;
}
```

Find

```
std::shared_ptr<Symbol> find(const std::string &name, size_t from = 0) const {
      if (from >= _contexts.size())
        return nullptr;
      for (size_t ix = _contexts.size() - from; ix > 0; ix--) {
        context_type &ctx = *_contexts[ix - 1];
        auto it = ctx.find(name);
        if (it != ctx.end())
          return it->second; // symbol data
      }
      return nullptr;
}
```

Tabela de símbolos

Scopes – Exemplo

```
#include <iostream>

int main() {
  int out = 1;
  int in = 3;
  {
    int in = 2;
    std::cout << "in  = " << in  << std::endl;
    std::cout << "out = " << out << std::endl;
  }
  out = 4;
  std::cout << "in  = " << in  << std::endl;
  std::cout << "out = " << out << std::endl;
  return 0;
}
```

- Output do programa?

Tabela de símbolos

Scopes: como geri-los?

- Criar uma pilha de scopes
  - push(): cria um novo scope
  - pop(): apaga o scope actual
- Uma tabela de símbolos por cada scope
- Procurar o símbolo em cada tabela, por ordem inversa de scope

Exercício:

- Estudar/inspecionar a tabela de símbolos da CDK: `/usr/include/cdk/symbol_table.h`
- Estudar/inspecionar o `symbol.h` do compilador `og`

Outline

Sistemas de tipos

Tipos de linguagens:

- Tipificação estática: C, Java, C++
  - Verificação em tempo de compilação
  - Geração de código mais eficiente
- Tipificação dinâmica: Common Lisp, Python
- Não tipificada: código máquina

Sistemas de tipos

Tipos de linguagens:

- Tipificação estática: C, Java, C++
  - Verificação em tempo de compilação
  - Geração de código mais eficiente
- Tipificação dinâmica: Common Lisp, Python
- Não tipificada: código máquina

Validações relativas aos tipos:

- Verificação de regras de consistência e promoções entre tipos
- Representação dos tipos criados pelo programador (`typedef`)
- Identificação dos métodos em caso de polimorfismo e herança

Consistência de tipos

O compilador deve verificar se os tipos das variáveis numa expressão são compatíveis

- Ex: não é possível adicionar um valor inteiro a um booleano

Adicionalmente, devem existir regras de conversão entre tipos

- Ex: int pode ser convertido para float

Inferência de tipos

Em sistemas de tipo estáticos, é possível ter inferência de tipos em tempo de compilação

Inferência de tipos

Em sistemas de tipo estáticos, é possível ter inferência de tipos em tempo de compilação

Exemplo:

- Templates em C++, onde a representação intermédia é instanciada para o tipo concreto

Inferência de tipos

Em sistemas de tipo estáticos, é possível ter inferência de tipos em tempo de compilação

Exemplo:

- Templates em C++, onde a representação intermédia é instanciada para o tipo concreto
- A leitura ou alocação de memória no projecto da cadeira, no contexto de uma atribuição

Inferência de tipos

Representação de tipos de informação no projeto

- A representação dos tipos de informação encontra-se presente no projeto através dos nós *tipificados* do AST:
  - `typed_node`, `expression_node`, `lvalue_node`, ...
- O tipo de cada nó é definido:
  - Pelo parser em alguns casos (por exemplo nas declarações)
  - Pelo analisador sintático na maioria dos casos
- A CDK fornece:
  - Os seguintes tipos básicos: `basic_type`, `primitive_type`, `reference_type`, `functional_type` e `structured_type`
  - funções que simplificam a instanciação dos tipos
    NOTA: Os tipos não são nós da AST, mas atributos que caraterizam o tamanho em memória das entidades

Exercício:

- Estudar/inspecionar os tipos da CDK: `/usr/include/types/`
- Estudar/inspecionar inferência de tipos no `ast/type_checker.cpp` do compilador `og`

Validação de tipos

Declaração tem o objectivo de criar uma ligação entre o identificador e o tipo, com as seguintes características:

- **Existência**: a ligação deve existir antes do uso do identificador
- **Unicidade**: a ligação é exclusiva. O identificador não pode estar associado a tipos diferentes dentro do mesmo scope
- **Âmbito**: a ligação permanece válida durante todo o scope

Validação de tipos

Declaração tem o objectivo de criar uma ligação entre o identificador e o tipo, com as seguintes características:

- **Existência**: a ligação deve existir antes do uso do identificador
- **Unicidade**: a ligação é exclusiva. O identificador não pode estar associado a tipos diferentes dentro do mesmo scope
- **Âmbito**: a ligação permanece válida durante todo o scope

O compilador deve portanto:

- Registar o tipo atribuído a cada identificador
- Verificar que foi declarado uma única vez no scope actual
- Verificar em cada ocorrência de um identificador que este foi previamente declarado

Validação de tipos

Relação com o projecto – exemplos

- $expr_1 + expr_2$: tipo de cada expressão pode ser int/pointer/real

Validação de tipos

Relação com o projecto – exemplos

- $expr_1 + expr_2$: tipo de cada expressão pode ser int/pointer/real
- $expr_1$ && $expr_2$: tipo de ambas as expressões deve ser int

Validação de tipos

Relação com o projecto – exemplos

- $expr_1 + expr_2$: tipo de cada expressão pode ser int/pointer/real
- $expr_1$ && $expr_2$: tipo de ambas as expressões deve ser int
- $expr_1 < expr_2$: tipo de ambas as expressões deve ser int ou real

Validação de tipos

Relação com o projecto – exemplos

- $expr_1 + expr_2$: tipo de cada expressão pode ser int/pointer/real
- $expr_1$ && $expr_2$: tipo de ambas as expressões deve ser int
- $expr_1 < expr_2$: tipo de ambas as expressões deve ser int ou real
- Chamada de funções:
  - Obter lista de parâmetros e tipo de resultado:
    - Em OG, esta informação é retirada da tabela de símbolos
    - Em L22, esta informação está codificada no tipo funcional da expressão
  - Validar argumentos da chamada
  - Validar tipo do resultado

Validação de tipos

Relação com o projecto – exemplos

- $expr_1 + expr_2$: tipo de cada expressão pode ser int/pointer/real
- $expr_1$ && $expr_2$: tipo de ambas as expressões deve ser int
- $expr_1 < expr_2$: tipo de ambas as expressões deve ser int ou real
- Chamada de funções:
  - Obter lista de parâmetros e tipo de resultado:
    - Em OG, esta informação é retirada da tabela de símbolos
    - Em L22, esta informação está codificada no tipo funcional da expressão
  - Validar argumentos da chamada
  - Validar tipo do resultado
- Indexação de ponteiros:
  - A base deve ser do tipo pointer e o índice do tipo int

Validação de tipos

Relação com o projecto – exemplos

- $expr_1 + expr_2$: tipo de cada expressão pode ser int/pointer/real
- $expr_1$ && $expr_2$: tipo de ambas as expressões deve ser int
- $expr_1 < expr_2$: tipo de ambas as expressões deve ser int ou real
- Chamada de funções:
  - Obter lista de parâmetros e tipo de resultado:
    - Em OG, esta informação é retirada da tabela de símbolos
    - Em L22, esta informação está codificada no tipo funcional da expressão
  - Validar argumentos da chamada
  - Validar tipo do resultado
- Indexação de ponteiros:
  - A base deve ser do tipo pointer e o índice do tipo int

Exercício:

- Estudar/inspecionar validação de tipos no `ast/type_checker.cpp` do compilador `og` (diferente nas funções a L22)

Validação de tipos

Relação com o projecto – Padrão visitor

- A verificação semântica do AST no projeto não é realizada de uma vez numa única passagem
- Uma instância do `type_checker` é criada pelo gerador de código sempre que é preciso realizar uma verificação com recurso à macro `ASSERT_SAFE_EXPRESSIONS`

```
#define CHECK_TYPES(compiler, symtab, node) { \
  try { \
    simple::type_checker checker(compiler, symtab, this); \
    (node)->accept(&checker, 0); \
  } \
  catch (const std::string &problem) { \
    std::cerr << (node)->lineno() << ": " << problem << std::endl; \
    return; \
  } \
}

#define ASSERT_SAFE_EXPRESSIONS CHECK_TYPES(_compiler, _symtab, node)
```

Outline

Type_checker.cpp

Ver exercício:

https://www.hlt.inesc-id.pt/~david/w3/pt/index.php/Semantic_Analysis/The_Tiny_language:_semantic_analysis_example_and_C_generation

Questões?

Dúvidas?