Apontamentos da disciplina de **Compilação**

Salvador Abreu

Ano lectivo 2002/03

$Id: apontamentos-comp.tex,v 1.3 2002/12/18 23:40:36 spa Exp $

Índice

1 Objectivos

É objectivo fundamental da disciplina de Compilação *familiarizar os estudantes com o desenvolvimento do processo de compilação*, ou tradução automática entre linguagens de programação, assim como com as *técnicas* e algumas *ferramentas* utilizadas para atingir este objectivo.

Outro objectivo é o desenvolvimento da aptidão para construir pequenos processadores de linguagens especializadas, utilizando um subconjunto das ferramentas de construção de compiladores, nomeadamente as promeiras fases, de análise lexical e sintactica.

O resultado final da componente prática da disciplina é um compilador para uma linguagem expressamente desenvolvida para esta finalidade, o TPL-03. Este compilador será individualizado por grupo, pois cada um deverá desenvolver, para além dum compilador que obedece a uma especificação comum, extensões diversificadas.

2 Motivação e enquadramento da disciplina

A disciplina de Compilação constitui um dos pontos focais para onde convergem de forma mais evidente, todo um conjunto de conhecimentos e técnicas que os estudantes assimilam ao longo dum curso de Licenciatura em Engenharia Informática. Com efeito, nesta disciplina são requeridos conhecimentos operacionais das seguintes áreas disciplinares:

- Programação,
- Algoritmos e estruturas de dados,
- Arquitectura de computadores,
- Linguagens formais.

Também se exercita a capacidade de realizar um projecto, já significativamente complexo, em equipa, ao longo dum semestre inteiro.

2.1 Precedências

A disciplina de Compilação tem como precedências recomendadas*explícitas* as seguintes disciplinas, todas elas existentes em pontos anteriores do plano de estudos recomendado da Licenciatura em Engenharia Informática:

- **Arquitectura de Computadores**. Esta disciplina do 1º semestre do 2º ano introduz os estudantes aos conceitos de organização de sistemas informáticos, do ponto de vista material, mas com uma grande ênfase na interface “hardware/software”. É apresentada uma arquitectura representativa (o MIPS) e são estudadas sucessivas implementações desta arquitectura. O estudo destas incide nos pontos de vista organizacional e da análise do impacto no desempenho. Em particular, são abordados vários aspectos tais como a organização da *arquitectura de instruções*, os sistemas de *memória virtual* assim como organizações visando exclusivamente obter ganhos de desempenho como as memórias *cache* e as *pipelines*.

Ao completar o requerido para obter aprovação nesta disciplina, o estudante adquiriu conhecimentos de programação em Assembler MIPS e uma sensibilidade para o impacto, em termos de desempenho nas implementações modernas, de várias opções quer de organização dum sistema quer de técnicas de programação.

- **Linguagens Formais e Autómatos**. Uma disciplina do 1º semestre do 3º ano em que são introduzidos os conceitos de linguagem formal, gramática para descrever uma linguagem e reconhecedor duma linguagem. Nesta disciplina são cobertos formalismos e classes de linguagens fundamentais para a compreensão da disciplina de Compilação, nomeadamente:

  – as linguagens regulares, descritas por expressões regulares e para as quais se podem construir reconhecedores implementados com autómatos finitos;

  – dentro das linguagens livres de contexto, as que podem ser descritas por gramáticas LL(k), LR(k) e LALR(k), e os reconhecedores associados as estas, podendo ser implementados como autómatos de pilha com e sem retrocesso.

  Ao obter aprovação nesta disciplina, o estudante está apto a caracterizar uma linguagem em termos do tipo de reconhecedor adequado para a descrever. Também compreende as limitações de cada tipo de reconhecedor e sabe lidar com a resolução das inevitáveis situações de ambiguidade e redundância na especificação duma gramática para uma linguagem de programação.

- **Linguagens de Programação**. Nesta disciplina do 2º semestre do 3º ano (antecedente cronológico imediato da disciplina de Compilação), são apresentadas características de diversas linguagens, que se pretende serem representativas de diversos paradigmas de programação. Para além da descrição das linguagens, esta disciplina discute as técnicas necessárias à sua implementação eficaz, em termos de estruturas de suporte à execução dos programas (p/ex. a organização de *stack frames*, mecanismos para suportar linguagens com recursão e acessos a variáveis não-locais, gestão de memória).

  O aproveitamento nesta disciplina confere ao estudante a sensibilidade necessária para compreender o custo das diferentes opções tomadas quer na definição duma linguagem de programação quer na estratégia escolhida para apoiar a sua implementação.

Estas três disciplinas podem naturalmente ser encaradas como os "pilares de sustentação" da disciplina de Compilação, pois apresentam a maior parte dos conceitos fundamentais cuja utilização é requerida nesta última.

3 Programa detalhado

Segue-se o programa detalhado das aulas teóricas.

3.1 Introdução

Inicialmente o processo de compilação é apresentado, sendo os objectivos deste processo e da disciplina explicitados.

3.1.1 O processo de tradução de programas

São apresentados e discutidos os seguintes conceitos:

- Representação dum programa.
- Processador para manipulação dessa representação.
- O processo de compilação como uma “pipeline” de processos independentes.
- Processos auxiliares persistentes (p/ex. tabela de símbolos).

3.1.2 Equivalência entre representações dum mesmo programa

A preservação da equivalência semântica entre representações dum mesmo programa deve ser assegurada por qualquer fase dum compilador.

3.1.3 Plano das aulas seguintes

É apresentada a sequência de tópicos para as aulas teóricas assim como o calendário de provas: os prazos de entrega dos trabalhos e a data da avaliação escrita.

3.1.4 Bibliografia

1. Capítulo 1 da referência principal.

3.2 Análise Lexical

A análise lexical é o primeiro passo dum compilador típico: é responsável pela leitura da representação externa (textual) dum programa e pela construção duma representação equivalente, composta por símbolos dum alfabeto mais estruturado e próximo da ideia que se tem da linguagem descrita; a estes símbolos convenciona-se chamar "tokens".

3.2.1 Linguagens regulares: caracterização

Recapitulação de conceitos já conhecidos dos estudantes, por terem obtido aprovação à disciplina de Linguagens Formais e Autómatos.

1. Hierarquia de linguagens de Chomsky.
2. Classes de linguagens e reconhecedores associados.
3. Breve tipificação de contstruções que podem ou não ser descritas por linguagens regulares.

3.2.2 Formalismos para descrever linguagens regulares

É recordada a notação –já conhecida de disciplinas anteriores– utilizada para descrever uma linguagem regular.

1. Gramáticas regulares.
2. Expressões regulares.

3.2.3 Reconhecedores para linguagens regulares: Autómatos finitos

É estabelecida a ligação entre uma linguagem descrita intensionalmente, uma descrição por via duma gramática e um reconhecedor que aceite frases dessa linguagem.

1. Autómatos finitos e a sua interpretação como reconhecedores:
   (a) Estados.
   (b) Transições.
   (c) Entradas.
   (d) Estados aceitadores.
2. Autómatos finitos não determinísticos (NFAs).
3. Construção dum NFA a partir dum conjunto (disjunto) de expressões regulares.
4. Autómatos finitos determinísticos (DFAs).
5. Construção dum DFA equivalente a um NFA.
6. Minimização do número de estados dum DFA.

3.2.4 Ferramenta para produzir reconhecedores para linguagens regulares: JFLex

O **JFlex** constroi reconhecedores em Java para o conjunto de expressões regulares indicado no seu input. Estas estão anotadas com *acções semânticas*, que não são mais do que segmentos de programa em Java para serem executados quando a expressão correspondente for reconhecida.

O JFlex vem na linha de ferramentas semelhantes existentes no Unix, para as linguagens C e C++: o Lex ou o seu sucedâneo GNU, o FLex, assim como a ferramenta utilizada no livro: o JLex.

1. Sintaxe dos ficheiros `.lex`.
2. Idiossincrasias do JFlex:
   (a) Estados iniciais ou contextos e seus usos.
   (b) A construção de "lookahead" e seus usos.

3.2.5 Bibliografia

1. Capítulo 3 do "Dragon Book".
2. Capítulo 2 da referência principal.
3. Documentação "on-line" do JFlex.

3.3 Análise sintáctica

O passo seguinte consiste no reconhecimento de sequências de "tokens" como frases da linguagem que se pretende analisar. Em termos de ferramentas para construção de compiladores, este passo foi durante muito tempo considerado como o mais importante, tome-se por exemplo o testemunho do *nome* do YACC (Yet Another Compiler Compiler, "Mais um compilador de compiladores") o qual aparenta significar que um compilador "é" aquilo que o YACC produz.

Convenciona-se designar estes reconhecedores sintacticos como "parsers".

3.3.1 Linguagens livres de contexto: caracterização

Revisão de conhecimentos: este assunto foi explorado na disciplina de Linguagens Formais e Autómatos.

3.3.2 Formalismos para descrever linguagens livres de contexto e classes de gramáticas

1. Gramáticas LL(1).
2. Gramáticas SLR(0).
3. Gramáticas LALR(1).
4. Notação EBNF.

3.3.3 Reconhecedores para linguagens livres de contexto

1. Autómatos de pilha.
    - Formas normais aceitáveis para as gramáticas e métodos para as re-escrever.
    - "Top-down"
        - Com retrocesso.
        - Predictivos, baseados em gramáticas LL(1).
    - "Bottom-up"
        - "Operator-precedence".
        - LR e variantes (LALR).
2. "Parsers" escritos manualmente.
    - Formas normais aceitáveis para as gramáticas e métodos para as re-escrever.
    - Funções FIRST(), FOLLOW() e NULLABLE() aplicadas a símbolos não-terminais.
    - Regras para construir manualmente um parser "recursive descent" numa linguagem de programação imperativa.

3. Tratamento de erros sintacticos.

   O tratamento de erros é um dos aspectos mais importantes que diferenciam um tratamento teórico, como o que é dado na disciplina de Linguagens Formais e Autómatos, duma abordagem prática como a que se pretende que seja utilizada numa ferramenta para uso concreto, como um compilador.

   - Manutenção de informação lexical (ficheiro, linha, coluna).
   - Abordagens para parsers "top-down" e "bottom-up".
   - Símbolos de sincronização.

3.3.4 Ferramenta para produzir reconhecedores para linguagens LALR(1): CUP

Tal como o JLex, o **CUP** é um préprocessador que produz código Java com um reconhecedor, neste caso um parser para uma gramática LALR(1). O CUP tem um uso próximo do tradicional YACC (ou Bison), embora esteja orientado para a utilização com a linguagem Java.

Uma alternativa é a utilização dum gerador de parsers baseado em gramáticas LL(1), como é o caso do PCCTS (Purdue Compiler Construction Tool Set) ou do seu sucessor ANTLR, que combina a funcionalidade do JFlex/JLex com a do CUP. Dado o ANTLR construir "parsers" baseados em gramáticas LL, este origina menos situações de dúvida para os estudantes, perante ambiguidades na gramática. O ANTLR também oferece mecanismos para facilitar a construção da árvore abstracta (ver a secção 3.8 na página 19). Outra vantagem do ANTLR é a de este definir automáticamente métodos para efectuar travessias da APT.

3.3.5 Bibliografia

1. Capítulo 3 da referência principal.
2. Capítulo 4 do "Dragon Book".
3. Documentação "on-line" do CUP e URL `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.
4. Documentação "on-line" do ANTLR e URL `http://www.antlr.org/`.

3.4 A linguagem TPL-03

Para o desenvolvimento dos trabalhos práticos, utiliza-se uma linguagem designada por TPL-03 (*Trivial Programming Language 02*) na qual se encontram características diversas das linguagens de programação imperativas, que exercitam um leque alargado de situações.

3.5 Elementos lexicais

As convenções lexicais são as habituais.

3.5.1 Comentários

Um comentário começa com o caracter '**#**' e termina no fim da linha em que este ocorre.

3.5.2 Identificadores

Os identificadores têm a definição habitual, coincidindo com a da linguagem C. A linguagem TPL-03 é sensível às diferenças entre minúsculas e maiúsculas.

3.5.3 Palavras Reservadas

Normalmente, as palavras reservadas têm como definição a própria palavra, em minúsculas. Algumas palavras reservadas podem aceitar uma representação textual alternativa:

Terminal	Representação
AND	&&
OR	||
NOT	~
RETURN	^
COND	?
WHILE	*
ELSE	*

Atenção que estas variantes podem causar colisões com as restantes definições de símbolos terminais, pelo que o tratamento destas situações poderá ser efectuado com precaução, havendo várias abordagens possíveis nomeadamente a nível da análise sintactica.

3.5.4 Constantes (Literais)

O analisador lexical deverá reconhecer constantes (literais) dos 3 tipos base apresentados, nomeadamente:

Constantes inteiras (`INT_LIT`). São constantes inteiras decimais, expressas pela definição habitual. Só são contempladas as este nível as constantes positivas, ie. sem sinal.

Constantes de vírgula flutuante (`REAL_LIT`). Tal como as anteriores, estas seguem as convenções habituais. No entanto, deverão ser reconhecidos, por exemplo, valores nas seguintes formas: "`.8`", "`0.005`", "`123e+17`", "`1.5e2`", "`3e-5`" e "`.200284E3`".

Constantes booleanas (`BOOL_LIT`). As constantes booleanas, literalmente `true` e `false`.

3.6 Sintaxe

A linguagem TPL-03 é apresentada informalmente pela gramática das figuras 1 a 5 (ver páginas 14 a 17). Esta gramática já se encontra numa forma fácilmente adaptável para especificar como input para um gerador de parsers LALR(1) como o CUP. Por uma questão de legibilidade (e tipografia) a gramática foi repartida em várias secções.

```
program -> decls                          /* Símbolo inicial */

decls -> /* VAZIO */                     /* Lista de declarações */
       | decls decl

decl ->                                   /* Declaração dum nome: */
        ids '=' type                      /* Definição de tipo */
      | ids ':' type                      /* Variável, tipo explícito */
      | ids ':' type ':=' exp             /* Variável, tipo explícito, init */
      | ids ':' type '=' exp              /* Constante, tipo explícito */
      | ids          '=' exp              /* Constante, tipo implícito */

formals -> /* VAZIO */                   /* Lista de parâmetros formais */
         | formal_decl formals

formal_decl ->                            /* Parâmetro formal: */
              ids                         /* Tipo implícito */
            | ids ':' type                /* Tipo explícito */

ids -> ID                                 /* lista de identificadores */
     | ID ',' ids

op -> '+' | '-' | '*' | '/' | '%'         /* Os operadores */
    | AND | OR | NOT
    | '<' | '<=' | '==' | '!=' | '>=' | '>'
```

Figura 1: EBNF para a linguagem TPL-03 – Declarações

Algumas observações sobre a gramática da linguagem TPL-03:

- Um *programa* em TPL-03 consiste numa sequência de declarações.

- A linguagem tem inferência de tipos, pelo que as declarações de nomes poderão omitir o seu tipo.
- A identidade de tipos é estrutural, pelo que tipos anónimos ou com nomes diferentes podem ser considerados idênticos, desde que a sua estrutura coincida.
- O constructor de "tuplo anónimo" (o símbolo '`,`') pode ser utilizado para construir *expressões primárias* ou *restritas* (ver figura 4). Na versão aumentada da linguagem, estas podem encontrar-se à esquerda dum símbolo de afectação ('`:=`'). Na versão base tal não é permitido pelo que o não terminal "`primary`" parece ser inútil nesta gramática.
- As especificações de tipo compreendem agregados. Estes podem ser, nomeadamente, anónimos (*tuplos*) ou etiquetados. Esta última possibilidade é expressa pela última regra do símbolo não terminal `single_type`, como se pode ver na figura 2.
- A regra do não-terminal "`sexp`" que começa por "`CLASS`" destina-se a expressar constantes de tipos agregados heterogéneos, p/ex. `CLASS a: int, b: bool [ a := 3; b := true ].`

```
type ->                          /* -- ASSINATURA DE TIPO -- */
        single_type               /* Um só tipo (fim de lista) */
      | '(' type ')'              /* Agrupamento sintactico */
      | single_type ',' type      /* Tuplo de tipos (lista) */

single_type ->                   /* -- EXPRESSÃO DE TIPO -- */
        ID                        /* Identificador de tipo */
      | INT                       /* Inteiro */
      | REAL                      /* Vírgula flutuante */
      | BOOL                      /* Booleano */
      | VOID                      /* Void (ex. instruções de controle) */
      | type '->' type            /* Tipo funcional */
      | '[' exp ']' type          /* Tipo "Array" */
      | '{' formals '}'           /* Tipo agregado (classe) */
```

Figura 2: EBNF para a linguagem TPL-03 – Declarações de tipo

3.7 Notas sobre Semântica

- A execução do programa consiste numa activação da função `program`, função esta que deverá ser definida pelo programador e não tem argumentos nem valor de retorno (tipo vazio em ambos os casos).
- Não é necessário definir um nome antes de o usar, bastando para tal que este esteja definido no mesmo "bloco" (âmbito ou "scope"), mesmo que posteriormente ao uso.

```
exp -> sexp                              /* -- EXPRESSÃO -- */
     | sexp ',' exp
     | '(' exp ')'

sexp -> sexp OR sexp                       /* Operadores booleanos */
      | sexp AND sexp
      | NOT sexp

      | sexp '<' sexp                      /* Operadores de comparação */
      | sexp '<=' sexp
      | sexp '==' sexp
      | sexp '!=' sexp
      | sexp '>=' sexp
      | sexp '>' sexp

      | sexp '+' sexp                      /* Operadores aritméticos */
      | sexp '-' sexp
      | sexp '*' sexp
      | sexp '/' sexp
      | sexp '%' sexp
      | '-' sexp

      | sexp '.' ID                        /* Nomes qualificados */
      | sexp '[' exp ']'                   /* Referências a arrays */
      | sexp '(' exp ')'                   /* Aplicação funcional */
      | '@' '(' exp ')'                    /* Aplicação recursiva directa */

      | ID                                 /* Nome simples */

      | INT_LIT                            /* Constante inteira */
      | REAL_LIT                           /* Constante em vírgula flutuante */
      | BOOL_LIT                           /* Constante booleana */

      | '[' exp ']'                        /* Literal de array */
      | MAP '(' formals ')' '[' stats ']' /* Literal funcional */
      | MAP '(' formals ')' '->' type  /* Idem, com tipo explícito */
                            '[' stats ']'
      | CLASS '(' formals ')' '[' stats ']' /* Literal de classe */
```

Figura 3: EBNF para a linguagem TPL-03 – Expressões

```
prim -> ID
      | prim '.' ID
      | prim '[' exp ']'

primary -> prim
         | primary ',' prim
         | '(' primary ')'
```

Figura 4: EBNF para a linguagem TPL-03 – Expressões restritas

```
stats -> /* VAZIA */                    /* -- INSTRUÇÕES -- */
       | stat ';' stats
       | stat

stat -> decl                             /* Declaração */
      | prim ':=' exp                    /* Afectação *** ATENÇÃO *** */
      | prim '(' exp ')'                 /* Chamada de função */
      | RETURN exp                       /* Retorno de função */
      | BREAK                            /* Saída de ciclo */
      | COND '[' clauses ']'             /* Instrução condicional */
      | WHILE '[' clauses ']'            /* Instrução de ciclo condicional */
      | '[' stats ']'                    /* Agrupamento de instruções */

clauses -> exp '->' stats                /* Instrução com guarda */
         | exp '->' stats '|' clauses
         | exp '->' stats '|' ELSE '->' stats
```

Figura 5: EBNF para a linguagem TPL-03 – Instruções

- As definições que constituem o não-terminal `program` dum programa serão designadas como definições *globais*, pelo que são reconhecidas em todo o programa.
- A passagem de parâmetros é sempre efectuada por *valor*.

3.8 Sintaxe abstracta

A par da análise sintactica, pode-se fazer uma primeira parte do que se convenciona designar por *análise semântica* (ver a secção 3.10, na página 27). Esta consiste na construção duma representação mais apropriada para a análise do programa. Este processo será conduzido pela sintaxe análise sintactica, daí a designação "syntax-directed translation", usada no "Dragon Book".

3.8.1 Acções semânticas

Antes de especificar acções semânticas numa regra, é necessário atribuir um tipo a cada símbolo da gramática (terminal e não-terminal). Também é necessário dar um nome aos símbolos que se pretende que sejam utilizados na acção semântica.

É estudado um exemplo duma gramática para expressões aritméticas. Os valores associados a cada não-terminal ou terminal "com valor" funcionam como atributos *sintetizados* quando encarados numa óptica de gramática de atributos.

As acções semânticas associadas às regras da gramática usam-se para construir estruturas que se designam no caso geral por *árvores*, embora estas possam em certos casos ser bem mais simples do que uma árvore.

1. **Gramáticas LL (parser recursivo descendente)**.
   Função (do tipo dos valores calculados) que retorna o valor correspondente à produção.

   No caso de gramáticas re-escritas para evitar recursão esquerda, é necessário ter cuidado pois nem todos os valores estarão disponíveis: neste caso pode-se passar argumentos suplementares às funções que implementam os não-terminais auxiliares, como se de atributos herdados se tratasse.

2. **Gramáticas LR (parsers gerados pelo YACC, BISON ou CUP)**.
   Aqui não é necessário recorrer aos artifícios anteriores. Na acção semântica associada a cada regra (que deve aparecer após o fim da regra), a cada elemento constituinte corresponde uma *ordem* dentro da regra (1, 2...) que será usara para referir o *resultado* associado a esse elemento: no YACC ou no BISON usa-se a notação `$1`, `$2`, etc. O *resultado* correspondente à redução pela regra corrente é referido por `$$`. No caso do CUP usa-se um identificador explícito, introduzido à frente do símbolo da produção.

   Os parsers gerados pelo YACC, BISON ou CUP mantém uma *pilha* associada à sua execução, que é em tudo paralela à pilha do parser LR pelo que contém um valor por cada símbolo gramatical presente. Dentro duma regra pode-se aceder "legalmente" aos valores associados aos símbolos presentes na regra.

   Deve-se ter em conta o *tipo* de cada símbolo, para isso usam-se as instruções `%union` e `%type` ou `<variante>` do YACC.

3. **Uso das acções semânticas para fazer um interpretador**.
   É simples fazer um interpretador com acções semânticas, por exemplo para uma linguagem

"tipo calculadora" executa-se a acção necessária mal uma regra é usada para reduzir, poderiamos por exemplo ter a seguinte gramática para o YACC ou BISON:

```
expr: expr '+' expr     { $$ = $1 + $3; }
    | expr '*' expr     { $$ = $1 * $3; }
    | '(' expr ')'      { $$ = $2; }
    | NUM               { $$ = $1; }
```

Esta abordagem corresponde à situação de uso mais simples, em que se tomam *imediatamente* todas as acções pretendidas (ou seja, não fica nada por fazer.)

Caso se esteja a usar o CUP, a notação difere ligeiramente e poderia, para a mesma situação, ser dada pelo código:

```
expr: expr:a '+' expr:b     {: RESULT = a + b; :}
    | expr:a '*' expr:b     {: RESULT = a * b; :}
    | '(' expr:e ')'        {: RESULT = e; :}
    | NUM:n                 {: RESULT = n; :}
```

A notação do CUP é ligeiramente mais expressiva e segura, embora não permita fazer alguns "hacks" que o YACC permite (p/ex. usar a notação `$-1` para designar items da pilha fora do contexto da regra...)

3.8.2 Mecanismo: construção da Árvore Abstracta

Seria possível fazer *todos* os passos dum compilador directamente nas acções semânticas do parser, no entanto, essa abordagem (por ser demasiado monolítica) seria impraticável pois o compilador tornar-se-ia gigantesco e difícil de ler. Para além disto, e dependendo da linguagem, quereremos examinar partes do programa em análise *várias vezes* o que se tornaria impossível nesta situação.

Assim, em vez de fazer todas as operações conducentes à compilação directamente sobre a análise sintactica, pode-se construir uma representação do programa mais *conveniente*, a qual se designa por *árvore de sintaxe abstracta* ou simplesmente *árvore abstracta*.

Entende-se por *conveniente* uma sintaxe (ela própria susceptível de ser descrita por uma gramática) que descreve os mesmos programas que os da sintaxe concreta. Esta sintaxe pode ser ambígua pois não é usada para analisar um string (o programa) mas *já dispõe* dessa análise e só serve para descrever uma estrutura (a árvore abstracta) ela própria já construida.

A *sintaxe abstracta* é semelhante à sintaxe concreta, depois de remover todos os símbolos que só existem para desambiguar a linguagem, para benefício quer do programador quer do analisador sintactico.

Os tipos que ocorrem na sintaxe abstracta também são objecto duma simplificação: procura-se utilizar os mecanismos mais básicos sempre que façam sentido. Por exemplo, uma instrução `for` do C seria representada como uma instrução composta contendo uma afectação (a inicialização) e um `while` ao corpo do qual se acrescenta outra afectação (a re-inicialização.)

3.8.3 Convenções de construção da APT em Prolog

Deve-se associar a cada não-terminal ou produção um termo com functor principal distinto. A escolha duma representação específica caberá sempre ao implementador, não existindo uma forma preferencial. Por exemplo, para a gramática de expressões aritméticas usada na secção 3.8.1, poderiamos usar a representação para as regras apresentadas:

```
binop(add, E1, E2)
binop(mul, E1, E2)
constant(V)
```

Note-se que a terceira regra (`expr: '(' expr ')'`) não requer representação a nível da APT pois os parêntesis só servem para explicitar uma estrutura com apoio da sintaxe linear textual.

Em alternativa a esta representação, poderiamos ter uma em que o functor principal seja sempre dado pelo nome do não-terminal em consideração:

```
expr(binop(add, E1, E2))
expr(binop(mul, E1, E2))
expr(constant(V))
```

Finalmente, podemos recorrer a uma representação mais compacta que qualquer destas, em que o contexto duma expressão implica uma restrição sobre o domínio dos valores que pode assumir, i.e.:

```
add(E1, E2)
mul(E1, E2)
constant(V)
```

3.8.4 Convenções de construção da APT com classes Java

Deve ser construida uma hierarquia de classes em que:

- A relação de herança deverá descrever uma estrutura de floresta.
- As raízes desta (as "superclasses") corresponderão aos nomes dos não-terminais relevantes[1] da gramática.
- Os descendentes desses nós (as "subclasses") corresponderão às diversas produções existentes na gramática para cada não-terminal.

[1]Entende-se por não-terminal *relevante* aquele que se prentede preservar na sintaxe abstracta, omitem-se nomeadamente aqueles que são introduzidos para desambiguar a gramática aos olhos do gerador de "parsers".

3.8.5 Notas sobre convenções de uso de tipos e nomes em C

Embora esta disciplina utilize o Prolog e o Java como linguagens de implementação, é comum ter de lidar com o C nesse papel, nomeadamente quando se trata de programas existentes que se pretende modificar. Por isso apresentam-se algumas recomendações sobre a forma de estruturar programas em C para uso com "parsers" produzidos pelo YACC/Bison e analizadores lexicais produzidos pelo Lex/Flex.

Quando se está a construir um compilador em que a linguagem de apoio é o C, convém obedecer a convenções por forma a facilitar o desenvolvimento modular e disciplinado do compilador. Seguem-se possíveis regras a observar para facilitar a programação:

1. Qualquer árvore pode ser descrita por uma gramática.
2. Os tipos usados numa árvore têm sempre um `typedef` associado, que corresponde a um símbolo da gramática.
3. Cada `typedef` define um apontador a uma `struct` relacionada: o nome da `struct` termina com um `_` e não é nunca mais usado excepto na declaração do `typedef`.
4. Cada `struct` contém dois campos:
   - Um `enum` com tipo anónimo, chamado `kind` cujos valores indicam qual das variantes é usada numa determinada instância. As variantes correspondem bi-univocamente às produções desse símbolo na gramática.
   - Uma `union` chamada `u`, com um campo de nome e tipo adequado a cada produção.
5. Caso haja mais de um símbolo não-trivial (com valor) na regra, a `union` terá um componente que será uma outra `struct` em que ocorrem (apropriadamente tipados e nomeados) os valores dos símbolos em questão.
6. Se só houver um símbolo não-trivial, a `union` terá um componente que é directamente o valor desse símbolo.
7. Para cada tipo (classe) haverá uma função *constructora* que inicializa todos os campos. Toda a alocação de memória é feita nestas funções.
8. Cada módulo (ficheiro `.h`) utilizará um prefixo único para todos os símbolos que define.
9. Depois do prefixo, todos os nomes de:
   - `typedefs` começam com uma minúscula.
   - constructores começam com uma maiúscula.
   - átomos de `enum` com uma minúscula.
   - nomes de variantes com uma minúscula.

3.8.6 Bibliografia

1. Capítulo 4 da referência principal.

3.9 Dicionários (tabelas de símbolos)

Uma tabela de símbolos eficaz e adaptada aos usos que pretendemos fazer num determinado compilador é um dos componentes mais importantes deste, pois uma concepção apropriada conferirá ao código (da tabela de símbolos) um elevado grau de re-utilizabilidade. Assim sendo, este assunto tem um destaque especial pois é importante que seja construída uma solução flexível e eficaz.

Os estudantes já tiveram contacto –na disciplina de Estruturas de Dados, do 1º semestre do 2º ano– com as técnicas que são aqui utilizadas, nomeadamente as "hash tables" e outras estruturas de dicionário. Trata-se portanto duma *aplicação* desses conceitos, que irá pôr à prova o seu domínio destas técnicas.

3.9.1 Definição formal do Tipo Abstracto de Dados "Dicionário"

- Mapeamento $\mathsf{dic} : \{\mathsf{Chave}\} \mapsto \{\mathsf{Valor}\}$.
- Operações sobre um dicionário.

3.9.2 Composição de dicionários

Organização dum dicionário para responder às operações:

- Inserção de par chave/valor,
- Consulta de par chave/valor,
- Listagem do conteúdo dum dicionário,
- União hierárquica de dois dicionários (com efeito de ocultação).

A última operação sobre dicionários será necessária para implementar linguagens em que exista o conceito de blocos de definições imbricados.

3.9.3 Dicionários funcionais

Representação dum dicionário como uma expressão. Implementação das funções de acesso ao dicionário como criação duma nova estrutura (organização funcional). Partilha de informação entre dicionários para conservação de memória.

3.9.4 Dicionários lógicos

A representação dum dicionário em Prolog poderá ser efectuada como factos que são manipulados com os predicados "built-in" `assertz/1` e `retract/1` ou como um termo incompleto. Dada a relativa ineficiência da primeira abordagem, conjugada com o benefício da variável lógica que caracteriza a segunda, iremos preferir a segunda.

Uma implementação das funções de acesso ao dicionário deverá estruturar-se como variantes do predicado `member/2`; considere-se por exemplo a função de "obtenção do valor associado a uma chave", dada pelo predicado `lookup/3`:

```
lookup(DICT, _, _) :- var(DICT), !, fail.
lookup([K=V|_], K, V).
lookup([_|DICT], K, V) :- lookup(DICT, K, V).
```

Depreende-se que um dicionário é aqui representado como uma lista "aberta", ie. com a sua cauda livre. Esta escolha permite expandir a estrutura de dados dum dicionário com novas associações, como se pode ver na seguinte definição para o predicado "inserir um valor associado a uma chave":

```
insert(DICT, K, V) :- var(DICT), !, DICT=[K=V|_].
insert([K=_|_], K, _) :- !, fail.
insert([_|DICT], K, V) :- insert(DICT, K, V).
```

Representações alternativas podem ser utilizadas, normalmente visando uma maior eficiência das operações mais frequentes (o `lookup` por exemplo). Em particular, poderá ser utilizada uma organização em árvore ou em "trie"; o que importa é manter a funcionalidade dos predicados `insert` e `lookup`.

O problema da composição de dicionários lógicos tem muitas soluções possíveis. Pode-se por exemplo utilizar a seguinte abordagem:

1. Um dicionário armazena as suas chaves como um termo especial, distinto dos outros. Por exemplo `id(CHAVE)` em vez de simplesmente `CHAVE`.

2. Se um dicionário, digamos `DI` (para Dicionário *Interior*) estiver contido num outro dicionário `DE` (para Dicionário *Exterior*) iremos proceder da seguinte forma:

   - Insere-se em `DI` uma entrada com chave, p/ex. `up` e valor `DE`, por forma a referir `DE` em `DI`.
   - Insere-se em `DE` uma entrada com chave `down` ou `down(N)` em que `N` é um inteiro distinto para todas as instâncias de `DI`. O objectivo é permitir que `DE` também tenha conhecimento de `DI`.
   - Ao fazer uma pesquisa (um `lookup/3`), deve-se procurar em `DE` após ter falhada uma busca em `DI`.

3.9.5 Dicionários modificáveis

Importa recordar as técnicas de representação eficiente de dicionários quando representados em linguagens imperativas (ie. com afectação). Recordam-se aqui estruturas de dados como as Hash Tables, as Árvores Binárias de Pesquisa, as Árvores Equilibradas ou Semi-Equilibradas, entre outras.

3.9.6 Bibliografia

1. Secção 5.1 da referência principal.

2. *The Java Language Reference*.
3. Apontamentos

3.10 Análise semântica (Nomes e Tipos)

A *análise semântica* já foi introduzida no contexto da análise sintactica (ver a secção 3.8 na página 19), sob a forma de "syntax-directed translation", ie. um processo que pode ser conduzido a par da análise sintactica propriamente dita. O que é aqui apresentado é um processo complementar, que pode ser efectuado sobre a *árvore abstracta*, e portanto já independentemente do processo de análise sintactica propriamente dito.

O processo aqui descrito consiste informalmente em "localizar aquilo que os nomes representam" e "completar e validar o uso de nomes e tipos", ou seja:

- Um nome é um *referente*, ie. um dispositivo que serve para designar algo.
- O processo de *análise de nomes* consiste na *desreferênciação* desses referentes, ie. a identificação do objecto referido.
- O processo de *análise de tipos* consiste no preenchimento completo da informação de *tipo* em todos os locais da árvore abstracta.

Habitualmente os processos de análise de nomes e tipos são feitos simultâneamente pois são mútuamente necessários.

3.10.1 Tabelas de símbolos e árvore abstracta

No processo de análise semântica pode-se trabalhar directamente sobre a árvore abstracta, em conjugação com a tabela de símbolos (que será por vezes designada como *contexto* ou "environment").

Alternativamente, pode-se tomar a perspectiva de desenvolver a tabela de símbolos em si, no pressuposto de os nomes que já lá estão terem valores que correspondem a partes da árvore abstracta. Isto significa percorrer a APT, transformando-a numa estrutura de dados centrada numa tabela de símbolos.

3.10.2 Análise de nomes

Como já foi aqui referido, a análise de nomes consiste na localização da definição correspondente a cada uso dum nome. Isto poderá traduzir-se na substituição do uso do nome por uma referência explícita e directa ao objecto referido.

Neste passo:

- São detectadas situações ilegais como:
  - Uso de nomes para os quais não existe definição.
  - Definições múltiplas para o mesmo nome.
- Qualquer referência a um nome é "desreferenciada", i.e. associa-se-lhe o valor presente no dicionário a que corresponde a definição aplicável.

3.10.3 Análise de tipos

A análise de tipos consiste:

- Na localização de todos os nós da árvore abstracta que têm ou necessitam de informação de tipo.
- No preenchimento da informação de tipo, caso esta seja omissa: o TPL-03 é uma linguagem com inferência de tipos, pelo que esta informação poderá não existir explícitamente em determinados pontos da árvore.
- Na verificação de que o tipo de cada nó está correctamente preenchido e utilizado.

Serão detectadas situações de uso incorrecto de tipos, por exemplo pela inadequação do uso duma determinada operação aos tipos dos seus operandos.

3.10.4 Concretização da análise de nomes e tipos para a linguagem TPL-03

Aplicação à linguagem TPL-03 do processo de análise de nomes e de tipos, implementando o compilador como um programa em Prolog.

O processo de análise semântica consistirá num percurso da árvore abstracta. Dada a situação em TPL-03 que permite que um nome seja utilizado *antes* de ter sido declarado (ou até a omissão total de qualquer declaração), torna-se necessário precaver as situações em que a ocorrência de uso aparece antes da de definição. Tal situação pode causar problemas, nomeadamente, quando se permitem acessos não locais.

Assim, o problema pode ser resolvido fácilmente efectuando uma traversia em largura da árvore abstracta, pois esta garante que todas as definições existentes num determinado nível (definido como a profundidade na árvore constituida pelos blocos de declarações) têm as suas ocorrências de definição numa de duas situações, nenhuma delas problemática:

1. A ocorrência de definição aparece antes de qualquer ocorrência de uso. Neste caso não há dificuldade nenhuma.
2. A ocorrência de definição aparece depois duma ocorrência de uso. Neste caso, e dado estarmos a efectuar uma travessia em largura da APT, temos a garantia de que, caso não tenha havido ...

No processo de re-escrita da APT, pode-se assumir que a representação à saída utiliza dicionários para cada nó. Esta solução poderá ser adoptada caso se pretenda simplificar a fase de análise de tipos, pois bastará associar um atributo `type` a cada nó, que indique qual o seu tipo. Assim, o tipo dum determinado nó, chamemos-lhe `NÓ`, da APT' (a versão re-escrita da APT) poderá ser imediatamente determinado com a seguinte instrução:

```
...
lookup(NÓ, type, T),
...
```

Desta forma, e tomando como exemplo uma expressão aritmética a ser analisada, poderemos transformá-la num dicionário com os seguintes atributos:

```
class=CLASS
type=TYPE
constant=CONST
lval=LVAL
value=VALUE
...
```

Em que:

- `CLASS` indica que tipo de nó estamos a considerar (poderá ser algo parecido com os functores principais dos nós da árvore abstracta, p/ex. `op`, `lit`, etc. Note-se que aqui convém *diluir* algumas distinções que possam ocorrer na APT, dado que se pretende que a nova representação seja mais rica e uniforme, recorrendo a menos esquemas de representação diferentes que a APT.

- `TYPE` será uma *expressão de tipos* que denote o tipo do nó em questão. Poderá ser utilizada uma construção simples, gerada (e portanto validada) por exemplo pelo seguinte predicado Prolog:

  ```
  tpl_type(int).
  tpl_type(bool).
  tpl_type(void).
  tpl_type(array(int,T)) :- tpl_type(T).              % arrays
  tpl_type((T1,T2)) :- tpl_type(T1), tpl_type(T2).    % tuplos
  tpl_type(class(ST)) :- symbol_table(ST).            % record
  tpl_type(map(T1,T2)) :- tpl_type(T1), tpl_type(T2). % função
  ```

  A penúltima clausula serve para denotar os tipos “record” que deverão ser representados com um dicionário em que as chaves são os nomes dos campos e os valores associados serão expressões de tipo.

- `CONST`, caso esteja ligada (“bound”, ie. não deixada como variável livre) terá como interpretação indicar se a expressão em questão é constante (e portanto calculável em tempo de compilação, caso se deseje fazer isso, mais tarde). A escolha dos valores é arbitrária mas poderá ser um dos átomos true e false.

- `LVAL` é semelhante a CONST mas indica se a expressão tem um “endereço,” por forma a poder usá-la como parte esquerda duma afectação.

- `VALUE` será algo que depende de `TYPE` e, ao critério do implementador, poderá ser:

  – Algo muito próximo da APT (ie. um termo com functor principal que indica qual a operação e subtermos com as sub-expressões, ou o termo que denota logo um literal) ou:

- outra representação, p/ex. uma lista, que poderá usar uma symbol table...

3.10.5 Bibliografia

1. Capítulo 5 da referência principal.

3.11 Registos de activação

Problemática da representação de *variáveis locais*:

- Cada invocação duma determinada função requer a sua própria *instância* destas.
- Após o retorno de função estas variáveis locais deixam normalmente de ser necessárias.

Como se observa que, numa linguagem com um controle de fluxo linear, as activações e retornos de função ocorrem por uma ordem LIFO, usa-se a estrutura de dados apropriada para representar as activações: uma pilha.

Esta representação para as activações só deixa de ser aplicável no caso de existirem funções que retornem outras funções (ie. funções de ordem superior), como é o caso das linguagens funcionais como o CAML ou o Haskell. Com efeito, nesta situação pode ser necessário referir uma ligação feita no interior duma função que já terminou. Neste estudo não iremos contemplar a implementação de linguagens que tenham essa característica.

3.11.1 Registos de Activação ou *Stack frames*

Se considerarmos que uma pilha é um TAD com as operações `push` e `pop` pode-se considerar várias formas de implementação. No caso que nos interessa, convém observar que os inícios e fins de função contém um grande número destas operações, pelo que convém reduzir o trabalho em tempo de execução: assim, organiza-se o *stack* como um vector, auxiliado por um registo especial, o *stack pointer* ou `SP`.

Chama-se *registo de activação* ou *stack frame* à zona (contígua) da pilha em que se encontra a informação relativa a uma activação em particular.

- **O *frame pointer***.

  Quando se entra numa função, o espaço das variáveis locais (e outros temporários) ainda vai ser alocado, na pilha. Este espaço pode ser de dimensão variável. Daí que seja conveniente representar a base do registo de activação, que poderá ser uma referência simples para os valores que estão presentes à entrada: o endereço de retorno e os valores dos parâmetros.

  Designa-se por `FP` um registo cuja função é a de representar o valor de `SP` à entrada da função. Este registo nem sempre é explícitamente representado.[2]

- **Registos**.

  As arquitecturas modernas definem um número *relativamente* grande (na ordem dos 32) de *registos de uso geral*, que se podem considerar como memórias de acesso rápido. Assim, é benéfico que o compilador faça uso destes sempre que possível.

[2]Embora a dimensão instantânea da pilha possa variar, a base do registo de activação poderá sempre ser determinada em função do valor de `SP` e dum deslocamento, possívelmente variável em função do local em que ocorre dentro da função.

Os registos, sendo em número limitado, devem ser salvaguardados quando se chama uma função: a responsabilidade desta operação pode caber à função que chama ou à que é chamada (*caller-save* vs. *callee-save*). Esta decisão é objecto de convenção para um determinado compilador: não é habitualmente imposta pelo hardware.

- **Passagem de parâmetros**.

  Os parâmetros são típicamente alocados no stack, mas esta abordagem tem sido preterida a favor do uso de registos particulares para os parâmetros.

  Tal abordagem requer mais cuidados pois:

  – Pode não haver registos suficientes.
  – Os registos podem ter de ser re-utilizados se a função chamada chamar outras funções.

  Não tendo por obrigatório a alocação de espaço no stack, torna-se possível optimizar o uso deste recurso, condicionando a decisão de guardar um parâmetro no stack à determinação de certas propriedades, por exemplo a não invocação de funções suplementares.

  Também é possível fazer alocação de registos generalizada (inter-procedimental), embora se trate de técnicas mais avançadas.

- **Endereço de retorno**.

  Típicamente o endereço de retorno era guardado na pilha, esta era a situação normal nas arquitecturas antigas (instrução `call` que empilha `PC+4`). Actualmente, prefere-se a abordagem de guardar o endereço de retorno num registo e *só se necessário* guardá-lo na pilha, evitando-se assim acessos desnecessários à memória.

- **Variáveis locais no registo de activação**.

  Sempre que possível (dependendo nomeadamente da disponibilidade de registos), é de evitar o uso de memória para as variáveis e resultados de expressões locais. Esta regra só deverá ser quebrada nas seguintes situações:

  – A variável será passada por referência: necessita dum endereço.
  – A variável é acedida por um procedimento imbricado no actual. Esta situação é passível de mudar se houver alocação global de registos.
  – A variável é demasiado grande para caber num registo. Inclui arrays.
  – O registo usado para representar a variável precisa de ser utilizado para outro fim, por exemplo passagem de parâmetros.
  – Há demasiadas variáveis locais para o número de registos. Às variáveis nesta situação são ”transbordadas” (*spilled*) para o registo de activação.

- **Ligações estáticas**.

  Só é necessário manter esta informação quando a linguagem permitir procedimentos imbricados. Usa-se o mecanismo do *display* já conhecido do Pascal, por exemplo. Recorde-se que este mecanismo consiste na manutenção dum vector global de apontadores para o registo de activação mais recente, indexado pelo nível lexical dos procedimentos. Cada registo de activação contém então uma referência, para além do registo de activação que o criou, ao registo de activação anterior, do mesmo nível lexical.

3.11.2 Concretização para a linguagem TPL-03

A matéria descrita neste capítulo é aplicada à construção e uso de registos de activação para a linguagem TPL-03.

3.11.3 Bibliografia

1. Capítulo 6 da referência principal.

3.12 A máquina de pilha SiM-03

A arquitectura SiM-03 pretende oferecer um alvo ("*target*") simples para geração de código num compilador para linguagens de programação imperativas. O conjunto de instruções é reduzido ao mínimo e estas só incorporam uma instrução que requer um operando (a instrução `PUSH`); todas as outras vão buscar os dados sobre os quais operam à pilha e colocam o seu resultado (caso o haja) nesta.

É fornecido um "compilador" de SiM-03 para assembler de Intel 386 para os estudantes poderem experimentar executar o seu código. A implementação é feita usando uma simples expansão de macros em M4, à qual se junta uma livraria com funções básicas de I/O.

3.12.1 Organização

Registos

A máquina SiM-03 dispõe dos seguintes registos:

`PC` (*Program Counter*).
Este registo designa o endereço da *próxima* instrução a ser executada.

`SP` (*Stack Pointer*).
Este registo designa o último endereço de memória a ser utilizado para colocar algo na pilha. Convenciona-se que a pilha cresce para baixo, ie. é uma PDL ou *Push-Down List*.

`FP` (*Frame Pointer*).
Este registo designa o endereço de memória onde começa o registo de activação do procedimento activo. Poderá ser interpretado como um valor passado do `SP`.

`SR` (*Scratch Register*).
Este é um registo *temporário*, que poderá ser utilizado sempre que conveniente. Existem operações de manipulação deste, interagindo com a pilha.

Memória

A máquina SiM-03 dispõe de palavras de 32 bits, 4Gbytes de memória[3] endereçáveis como bytes ou como palavras (alinhadas), ou seja, tem endereços de 32 bits também.

As zonas de memória têm um uso convencionado, que é o seguinte:

- Endereços $\texttt{00000000}_{16}$ a $\texttt{7FFFFFFF}_{16}$: disponíveis para o programa "utilizador".
  - **`STATIC`**: os endereços correspondentes ao Megabyte mais baixo (ou seja entre $\texttt{00000000}_{16}$ e $\texttt{000FFFFF}_{16}$) estão reservados a dados estáticos, disponíveis para todo o programa.
  - **`CODE`**: os endereços baixos seguintes $\texttt{00100000}_{16}\ldots$ estão reservados para o *código*.

[3] A implementação `sim2c` da máquina SiM-03 não disponibiliza tanta memória, sendo que as zonas de memória `DATA`, `HEAP` e `STACK` dispõem cada uma de 1MB e a zona `CODE` não é acessível em leitura, pois é compilada para C.

  - **DATA**: os endereços a seguir ao código servem para armazenar dados dinâmicos, e entende-se que "crescem" para endereços mais altos.
  - **STACK**: os endereços altos (ie. para baixo do endereço $\texttt{7FFFFFFF}_{16}$) são reservados para a *pilha*, e são interpretados como "crescendo" para endereços mais baixos.
- Endereços $\texttt{80000000}_{16}$ a $\texttt{FFFFFFFF}_{16}$: reservados para o "sistema operativo".

3.12.2 Arquitectura de Instruções (ISA)

Implícitamente, cada instrução começa por incrementar o program counter, i.e. é como se a semântica de cada uma começasse por $\texttt{PC} \leftarrow \texttt{PC} + 4$. Todas as instruções, com a excepção da instrução `PUSH` obtém os seus parâmetros da pilha. Esta última instrução empilha um valor que é especificado pelo seu parâmetro.

Algumas instruções usam e modificam simultâneamente um determinado registo. Caso esse uso seja ambíguo, o registo em questão será rotulado com um subscrito *temporal* para designar em que instante é que o valor em questão deve ser considerado. Por exemplo, $\texttt{PC}_{\mathrm{T}+1}$ designa o registo `PC` no final da instrução enquanto $\texttt{SP}_{\mathrm{T}}$ designa o registo `SP` no início da instrução.

O quadro 1 (ver na página 36) apresenta as instruções da arquitectura SiM-03, indicando a sua sintaxe e semântica.

3.12.3 Uso

Os quadros 2 e 3 (ver nas páginas 37 e 38) apresentam alguns exemplos típicos de uso desta arquitectura, confrontando-os com o código em C equivalente.

3.12.4 Assembler de SiM-03

Para facilitar o uso do SiM-03, foi desenvolvido um "assembler" de SiM-03 para C, chamado `sim2c`. Este assembler consome um ficheiro em assembler de SiM-03 e produz como resultado um programa em C, pronto a ser compilado.

Convenções

O `sim2c` não tem opções de compilação. O seu input são instruções SiM-03, possívelmente com labels (dados pela sintaxe `LABEL:` no início duma linha) e comentários (tudo o que vai desde o caracter `#` até ao fim duma linha).

O `sim2c` é "case insensitive" relativamente aos nomes das instruções e das directivas, não o sendo para os nomes definidos nos labels.

Os campos das instruções são delimitados de forma livre, desde que haja espaço em branco. São suportadas as seguintes directivas, todas inspiradas na sintaxe tradicional dos "assemblers" do Unix:

Operação	**Semântica**
PUSH x	$\mathtt{SP} \leftarrow \mathtt{SP} - 4;\ \mathrm{MEM}[\mathtt{SP}] \leftarrow x$
LOCAL	$\mathrm{MEM}[\mathtt{SP}] \leftarrow \mathtt{FP} + \mathrm{MEM}[\mathtt{SP}]$
STACK	$\mathrm{MEM}[\mathtt{SP}] \leftarrow \mathtt{SP} + \mathrm{MEM}[\mathtt{SP}]$
POP	$\mathtt{SP} \leftarrow \mathtt{SP} + 4$
DUP	$\mathtt{SP} \leftarrow \mathtt{SP} - 4;\ \mathrm{MEM}[\mathtt{SP}] \leftarrow \mathrm{MEM}[\mathtt{SP} + 4]$
SWAP	$\mathrm{MEM}[\mathtt{SP}] \leftrightarrow \mathrm{MEM}[\mathtt{SP} + 4]$
LOAD	$\mathrm{MEM}[\mathtt{SP}]_{32} \leftarrow \mathrm{MEM}[\mathrm{MEM}[\mathtt{SP}]]$
LOAD_B	$\mathrm{MEM}[\mathtt{SP}]_{8} \leftarrow \mathrm{MEM}[\mathrm{MEM}[\mathtt{SP}]]$
STORE	$\mathrm{MEM}[\mathrm{MEM}[\mathtt{SP}]]_{32} \leftarrow \mathrm{MEM}[\mathtt{SP} + 4];\ \mathtt{SP} \leftarrow \mathtt{SP} + 8$
STORE_B	$\mathrm{MEM}[\mathrm{MEM}[\mathtt{SP}]]_{8} \leftarrow \mathrm{MEM}[\mathtt{SP} + 4];\ \mathtt{SP} \leftarrow \mathtt{SP} + 8$
ADD	$\mathrm{MEM}[\mathtt{SP} + 4] \leftarrow \mathrm{MEM}[\mathtt{SP} + 4] + \mathrm{MEM}[\mathtt{SP}];\ \mathtt{SP} \leftarrow \mathtt{SP} + 4$
SUB	$\mathrm{MEM}[\mathtt{SP} + 4] \leftarrow \mathrm{MEM}[\mathtt{SP} + 4] - \mathrm{MEM}[\mathtt{SP}];\ \mathtt{SP} \leftarrow \mathtt{SP} + 4$
MUL	$\mathrm{MEM}[\mathtt{SP} + 4] \leftarrow \mathrm{MEM}[\mathtt{SP} + 4] \times \mathrm{MEM}[\mathtt{SP}];\ \mathtt{SP} \leftarrow \mathtt{SP} + 4$
DIV	$\mathrm{MEM}[\mathtt{SP} + 4] \leftarrow \mathrm{MEM}[\mathtt{SP} + 4] / \mathrm{MEM}[\mathtt{SP}];\ \mathtt{SP} \leftarrow \mathtt{SP} + 4$
MOD	$\mathrm{MEM}[\mathtt{SP} + 4] \leftarrow \mathrm{MEM}[\mathtt{SP} + 4] \% \mathrm{MEM}[\mathtt{SP}];\ \mathtt{SP} \leftarrow \mathtt{SP} + 4$
SLT	**se** $\mathrm{MEM}[\mathtt{SP} + 4] < \mathrm{MEM}[\mathtt{SP}]$ **então** $\mathrm{MEM}[\mathtt{SP} + 4] \leftarrow 1$ **senão** $\mathrm{MEM}[\mathtt{SP} + 4] \leftarrow 0;\ \mathtt{SP} \leftarrow \mathtt{SP} + 4$
JUMP	$\mathtt{PC} \leftarrow \mathrm{MEM}[\mathtt{SP}];\ \mathtt{SP} \leftarrow \mathtt{SP} + 4$
CALL	$\mathtt{PC}_{\mathtt{T}+1} \leftarrow \mathrm{MEM}[\mathtt{SP}]_{\mathtt{T}};\ \mathrm{MEM}[\mathtt{SP}]_{\mathtt{T}+1} \leftarrow \mathtt{PC}_{\mathtt{T}}$
SKIPZ	**se** $\mathrm{MEM}[\mathtt{SP}] = 0$ **então** $\mathtt{PC} \leftarrow \mathtt{PC} + 4;\ \mathtt{SP} \leftarrow \mathtt{SP} + 4$
LINK	$\mathrm{MEM}[\mathtt{SP}]_{\mathtt{T}+1} \leftarrow \mathtt{FP}_{\mathtt{T}};\ \mathtt{FP}_{\mathtt{T}+1} \leftarrow \mathtt{SP}_{\mathtt{T}};\ \mathtt{SP}_{\mathtt{T}+1} \leftarrow \mathtt{SP}_{\mathtt{T}} - \mathrm{MEM}[\mathtt{SP}]_{\mathtt{T}}$
UNLINK	$\mathtt{SP}_{\mathtt{T}+1} \leftarrow \mathtt{FP}_{\mathtt{T}};\ \mathtt{FP}_{\mathtt{T}+1} \leftarrow \mathrm{MEM}[\mathtt{FP}_{\mathtt{T}}]$
DUP_SR	$\mathtt{SR} \leftarrow \mathrm{MEM}[\mathtt{SP}]$
PUSH_SR	$\mathtt{SP} \leftarrow \mathtt{SP} - 4;\ \mathrm{MEM}[\mathtt{SP}] \leftarrow \mathtt{SR}$

Quadro 1: Arquitectura de Instruções para a SiM-03

Código C e SiM-03	Observações

`a = b+4;`
 	
`PUSH b ; endereço de b`
`LOAD ; valor de b`
`PUSH 4 ; constante 4`
`ADD ; (b)+4`
`PUSH a ; endereço de a`
`STORE ; a = (b)+4`
	Afectação de variáveis globais: Considera-se que `a` e `b` são variáveis globais, com endereços absolutos.

`x = foobar (2, a+3);`
 	
`DUP ; espaço r.v.`
`PUSH 2 ; arg 1: 2`
`PUSH -8 ; arg 2: a+3`
`LOCAL ; endereço de a`
`LOAD ; valor de a`
`PUSH 3 ; +3`
`ADD ; (fim arg 2)`
`PUSH foobar ; end. função`
`CALL ; chama`
`POP ; salta arg 2`
`POP ; salta arg 1`
`PUSH x ; endereço de x`
`STORE ; guarda valor`
	Chamada de procedimento com retorno de valor: Note-se que a instrução `DUP` serve neste caso unicamente para reservar espaço para o valor de retorno. Note-se que os argumentos são empilhados por ordem inversa, por forma a que o último fique mais perto do topo da pilha. Também é de realçar que compete ao chamador desempilhar os argumentos. A variável `a` é uma variável local, cujo endereço é $\texttt{FP} - 8$.

Quadro 2: Exemplos de uso da SiM-03

Código C e SiM-03	Observações

`int plus (int a, int b) {`
`  return a+b;`
`}`
 	
--- 	
 	
`PUSH 0  ; zero variáveis locais`
`LINK  ; ...`
`PUSH 12  ; deslocamento de a`
`LOCAL  ; endereço de a`
`LOAD  ; valor de a`
`PUSH 8  ; deslocamento de b`
`LOCAL  ; endereço de b`
`LOAD  ; valor de b`
`ADD  ; a+b`
`PUSH 16  ; deslocamento de r.v.`
`LOCAL  ; endereço de r.v.`
`STORE  ; (r.v.) = a+b`
`UNLINK  ; desfaz stack frame`
`JUMP  ; retorna`
	Definição de procedimento com retorno de valor: note-se que o acesso aos parâmetros é feito pelas sequências "`PUSH xx`; `LOCAL`; `LOAD`". Também é de realçar que a "stack frame" é construída usando as instruções "`PUSH xx`; `LINK`" e desfeita usando as instruções "`UNLINK`; `JUMP`". O endereço do valor de retorno é dado pelo que seria o "0-ésimo" argumento, ie. uma palavra antes do primeiro argumento.

Quadro 3: Exemplos de uso da SiM-03

`.DATA` Esta directiva indica ao assembler que o que se segue deverá ser assemblado no "data segment", i.e. na zona de memória em que se pode escrever.

`.TEXT` Esta directiva antecede instruções que se pretende sejam executáveis.

`.WORD` Esta directiva inicializa uma palavra com o valor de 32 bits que tiver como argumento. Podem ser dados múltiplos valores que irão ser assemblados em endereços consecutivos.

`.SPACE` Esta directiva reserva *N palavras* em que *N* é o argumento, interpretado como um inteiro decimal. Corresponde a fazer `.WORD 0, 0, ..., 0` com *N* `0`s.

3.12.5 Bibliografia

1. Apontamentos sobre a arquitectura SiM-03.

3.13 Geração de código para a máquina SiM-03

Neste capítulo serão descritas algumas situações de geração de código para a SiM-03, em termos genéricos e para uma linguagem procedimental clássica. Ocasionalmente serão apresentadas situações que ocorram na linguagem TPL-03.

3.13.1 Esquema de geração de código SiM-03

Tratando-se duma simples máquina de pilha sem registos "general purpose", a problemática da geração de código encontra-se considerávelmente simplificada, ao ponto de já estarmos em condições de gerar o código objecto sem mais análises a efectuar sobre a representação que temos actualmente do programa (a árvore abstracta).

3.13.2 Bibliografia

1. Capítulo 1 da referência principal.

3.14 Geração de código intermédio

Em alternativa à geração de código para a arquitectura SiM-03, pode-se ter como objectivo a geração de código para arquitecturas reais. Se encararmos um compilador como uma sequência de fases (às vezes designados por "passos"), a produção de código após a árvore abstracta aparece em grande medida como independente da linguagem que se está a compilar.

Para potenciar a re-utilização de código (objectivo sempre desejável numa optica de Engenharia de Software), faz todo o sentido não criar geradores de código directamente ajustados para arquitecturas específicas (Intel 32, Power PC, Alpha, etc...) mas sim procurar uma representação intermédia "universal", suficientemente próxima das arquitecturas reais para que a tradução para estas seja muito simples mas susceptível de permitir que se efectuem transformações sobre o código intermédio.

3.14.1 Geração de código intermédio

A linguagem usada para a representação intermédia define programas cuja representação é, à semelhança da sintaxe abstracta, uma árvore.

1. **Representação Intermédia (RI)**.

   Um programa na RI será representado por uma estrutura de dados composta por *nós*.

   Classificaremos os nós da RI em duas categorias: os tipos para representar expressões e os tipos para representar as restantes instruções (*statements*).

   Os nós das árvores RI para representar expressões estão apresentados, duma forma independente da linguagem de implementação, no quadro 4 (ver página 41).

Nome	**Observações**
CONST(i)	Constante inteira `i`.
NAME(n)	Constante simbólica `n` (típicamente um endereço).
TEMP(t)	Temporário. Semelhante a um registo numa máquina real.
BINOP(o,e_1,e_2)	Aplicação da operação `o` aos operandos e_1 e e_2.
MEM(e)	O conteúdo da célula de memória no endereço `e`.
CALL(f,l)	Invocação do procedimento `f` com a lista de argumentos (expressões) `l`.
ESEQ(s,e)	Idêntico a `e`, mas depois de avaliar a instrução `s` (efeitos secundários).

Quadro 4: RI para expressões

   Para representar instruções que não expressões, utilizam-se os tipos descritos no quadro 5 (ver página 42).

   A RI aqui utilizada não é a única possibilidade, sendo viável propôr outras organizações, das quais o *3-address code* é um exemplo (ver o "Dragon Book").

Nome	Observações
MOVE(TEMP(t), e)	Avaliar `e` e guardar no temporário `t`.
MOVE(MEM(e1), e2)	Avaliar `e2` e guardar no endereço `e1`.
EXP(e)	Avaliar `e` (ignorando o resultado).
JUMP(e, l)	Avaliar `e` e indexar na lista `l` para saltar.
CJUMP(o,e1,e2,t,f)	Avaliar "`e1 o e2`", saltando para `t` se for 1 ou para `f` caso contrário.
SEQ(s1,s2)	Avaliar `s1` e depois `s2`.
LABEL(l)	Define a etiqueta `n`, para usar como `NAME(l)`. Não executa nada.

Quadro 5: RI para outras instruções

2. **Tradução para árvores**.
   O objectivo é traduzir um programa expresso como uma árvore abstracta (APT) para uma árvore em representação intermédia (RI).

   O problema da representação dos tipos específicos e programação das operações sobre estes, numa linguagem específica como o Java ou o C, poderá ter pormenores densos mas o princípio é simples.

   Trata-se de, para cada tipo de nó da APT, gerar nós para a árvore de RI. Levantam-se algumas questões relativas a certas construções que ocorrem na APT.

   (a) **Tipos de expressão**.
   Questão: será que um nó do tipo *expressão*$_{\text{APT}}$ na APT deverá ser representado por um nó do tipo *expressão*$_{\text{RI}}$ correspondente, para a RI?

   Depende do uso que se pretende fazer da expressão, se:

   - Para realmente ficar com o resultado (caso da expressão ocorrer no contexto de outra expressão).
   - Para avaliar a expressão mas ignorar o resultado (caso da expressão ocorrer numa sequência de instruções).
   - Para avaliar a expressão (de tipo booleano) como um condicional (caso da expressão ocorrer como condição numa instrução condicional)

   Estas variantes serão codificadas como subtipos (subclasses, caso a linguagem de implementação seja o Java) da expressão na RI.

   No caso da avaliação condicional, por exemplo a *condição* dum `if`, faz parte da expressão uma lista explícita dos *labels* das instruções para onde se deve saltar no caso da avaliação resultar em `true` e `false`. Costuma-se chamar a isto uma *patch list*: a ideia é que os labels podem ainda não ter sido declarados na altura em que se cria o nó da RI correspondente à expressão.

   Também é conveniente dispor de funções de conversão de RI entre os diversos subtipos, por forma a poder garantir que, num determinado contexto se poderá dispor da representação necessária, independentemente da representação inicial.

(b) **Acesso a variáveis**.
Estas são encaradas e representadas como endereços de memória. No caso de variáveis locais (pertencentes ao registo de activação da função que engloba a expressão em causa) a representação usada poderá ser da forma:

MEM(BINOP(+, TEMP(`fp`), CONST(`k`)))

em que `k` é o deslocamento dentro da stack frame para a variável em questão (ver a secção 3.11, na página 31).

É de realçar o uso do registo `fp` ("Frame Pointer") encapsulado numa expressão TEMP.

No caso do acesso a variáveis externas, em que temos de seguir os "links" estáticos entre registos de activação – possívelmente através dum "display" – esta expressão poderá ser mais complexa pois teremos de *substituir* a ocorrência de "TEMP(`fp`)" por uma expressão da forma:

MEM(BINOP(+, LABEL(`DISPLAY`), CONST(`SCOPE`)))

Note-se também que um nó MEM representa um *acesso à memória*, ou seja, irá traduzir-se por um LOAD ou um STORE conforme o contexto. A não-distinção a este nível permite uma maior simplicidade na geração de sub-árvores RI.

Nos casos de referências compostas (arrays ou records) o princípio é o mesmo, mas as expressões serão mais complexas pois terão de envolver o cálculo do endereço como uma base e um deslocamento. Teremos a este nível aquilo que é visível a nível da linguagem, em C: a aritmética de endereços.

3. **Declarações**.
Estas serão representadas essencialmente como *alocações de espaço* na stack frame.

As definições de valores iniciais para variáveis deverão resultar na emissão de instruções RI para afectar as zonas de memória correspondentes às variáveis em questão com os valores das expressões de inicialização.

4. **Declarações de função**.
Nas declarações de função será preciso produzir código específico para a entrada na função e código para a saída da mesma.

Chamam-se a estes troços de código respectivamente *prólogo* e *epílogo*. Assim, o código duma função será composto por:

(a) O prólogo:

i. Uma definição de label para a função.

ii. Um ajuste do stack pointer, suficiente para acomodar todas as variáveis locais e temporárias: reserva-se espaço para o novo registo de activação.

iii. Instruções para guardar o contexto que for necessário no registo de activação recém-criado. Inclui os seguintes:

- Instruções para guardar os registos que devem ser guardados pelo chamado (*callee-saved*), incluindo o endereço para o valor de retorno da função.
- Instruções para ajustar o "static link", nomeadamente a ligação ao registo de activação anterior por via do registo `fp` e o ajuste do "display".

(b) O corpo da função.

(c) O epílogo:

i. Uma instrução para colocar o valor de retorno da função no local convencionado.
ii. Instruções para repôr os registos guardados à entrada da função (os "callee-saved").
iii. Instruções para repôr o "static link", incluindo ajustes ao "display" e ao apontador para a base do registo de activação: o `fp`.
iv. Instruções para repôr o "stack pointer".
v. Uma instrução de salto para efectivar o regresso à função chamadora.

No prólogo e no epílogo, algumas secções poderão ser vazias, dependendo do que a função fizer (nomeadamente, se não chamar mais nenhuma função).

3.14.2 Bibliografia

1. Capítulo 7 da referência principal.
2. Capítulo 8 do "Dragon book".

3.15 Blocos básicos e traços

A RI é concebida para ser fácil de gerar a partir da APT, sendo próxima duma arquitectura real. No entanto, é possível efectuar sobre a RI algumas transformações visando aproximar esta ainda mais da arquitectura pretendida. Alguns exemplos deste tipo de situação:

- O CJUMP salta para dois endereços mas as máquinas reais têm saltos condicionais que só vão para um.
- Os nós ESEQ são problemáticos pois tornam o resultado dependente da ordem de avaliação.
- O mesmo se pode dizer acerca dos nós CALL dentro de expressões.
- Nós CALL dentro de outros CALLs vão dar problemas se a convenção de chamada precisar de colocar os argumentos em registos específicos.

Para resolver estes problemas vamos fazer várias transformações sobre a RI:

1. Transformar a árvore em RI numa lista de *árvores canónicas* sem SEQ nem ESEQs (a sequência será implícitamente representada pela ordem dos elementos da lista).
2. Agrupar estas árvores em *blocos básicos*: sequências que não contenham saltos nem etiquetas *internos*.
3. Ordenar os blocos básicos em *traços* de tal forma que um bloco básico que termine num CJUMP seja imediatamente seguido pelo bloco básico referido pelo endereço `false` do anterior.

A implementação dos algoritmos que irão reconhecer estas situações e efectuar as transformações indicadas poderá ser feita directamente sobre o compilador que vem sido construido em Java. No entanto, e dada a natureza do problema, poderá ser interessante explorar a via de programar estes algoritmos numa linguagem mais apropriada como por exemplo:

- Uma linguagem funcional tipada, por exemplo o CAML ou o Haskell. Esta abordagem é viável pois os estudantes já tiveram contacto com pelo menos uma das duas.
- Uma linguagem de Programação em Lógica como o Prolog. Esta escolha será deveras a mais prometedora pois as facilidades de "pattern matching" oferecidas pelo mecanismo de Unificação do Prolog permitem resolver muitos dos problemas seguidamente enunciados de forma muito simples.

Independentemente de qual a segunda linguagem de programação escolhida, será necessário definir uma *representação externa* para, pelo menos, a APT. Esta será produzida pelo código Java existente e consumida por esta fase do compilador, escrita na outra linguagem.

3.15.1 Árvores canónicas

O objectivo é retirar os nós SEQ e ESEQ. Também se reorganiza a RI por forma a que o nó pai de cada CALL seja um EXP ou um MOVE(TEMP(t, ...)).

1. **Remoção de SEQ e ESEQ**
   Ideia: re-escrever a RI por forma a retirar todos os SEQ e ESEQ do interior da árvore, fazendo-os aparecer como se tratasse de constructores de listas. Assim, um ESEQ nunca será descendente de algo que não outro ESEQ e um SEQ também.

   A abordagem será por reconhecimento de padrões (*pattern matching*) sobre a forma da RI original. As re-escritas estão indicadas no quadro 6 (ver página 46).

	Original	**Tansformada**	**Observações**
1	ESEQ(s_1,ESEQ(s_2,e))	ESEQ(SEQ(s_1,s_2),e)	
2	BINOP(op,ESEQ(s,e_1),e_2) MEM(ESEQ(s,e)) JUMP(ESEQ(s,e)) CJUMP(op,ESEQ(s,e_1),e_2,t,f)	ESEQ(s,BINOP(op,e_1,e_2)) ESEQ(s,MEM(e)) SEQ(s,JUMP(e)) SEQ(s,CJUMP(op,e_1,e_2,t,f))	
3	BINOP(op,e_1,ESEQ(s,e_2)) CJUMP(op,e_1,ESEQ(s,e_2),l_1,l_2)	ESEQ(MOVE(TEMPt,e_1), ESEQ(s,BINOP(op,TEMP t,e_2))) SEQ(MOVE(TEMP t,e_1), SEQ(s,CJUMP(op,TEMP t,e_2,l_1,l_2)))	Novo temporário:t.
4	BINOP(op,e_1,ESEQ(s,e_2)) CJUMP(op,e_1,ESEQ(s,e_2),l_1,l_2)	ESEQ(s,BINOP(op,e_1,e_2)) SEQ(s,CJUMP(op,e_1,e_2,l_1,l_2))	Só se s e e_1 comutarem.

Quadro 6: Migração dos nós SEQ e ESEQ

   Note-se que o caso 4 é uma optimização relativamente ao 3, que só será possível se pudermos *garantir* por inspecção da RI que a comutatividade é aplicável.

2. **Re-escrita**
   O objectivo será ficar com uma função que reagrupe a RI de maneira a ficar sob a forma duma lista de pares (instrução, expressão).

3. **Elevação dos CALL**
   Para garantir que nunca ocorre um CALL dentro doutro, temos de re-escrever um CALL com outros CALLs lá dentro sob a forma dum ESEQ em que os CALLs interiores são primeiro avaliados, ficando o seu resultado armazenado num temporário novo. O CALL de topo tomará assim como lista de argumentos os originais que não eram CALLs, sendo os CALLs substituidos pelos temporários recém-criados.

3.15.2 Saltos condicionais

A ideia vai ser re-ordenar as sequências de instruções construídas anteriormente por forma a garantir que uma sequência que termine num CJUMP seja imediatamente seguida pela sequência que começa pelo label referido no `false` do CJUMP.

1. **Blocos básicos**
   Faz-se a análise de *fluxo de controle* do programa: olha-se para a RI disponível, ignorando tudo o que não forem labels ou saltos.

   Um *bloco básico* é uma sequência de instruções em que:

   - A primeira é um LABEL
   - A última é um JUMP ou um CJUMP
   - Não há nenhum outro LABEL, JUMP ou CJUMP

   A divisão em blocos básicos pode resultar em qualquer ordem para o programa que o significado será o mesmo.

2. **Traços**
   Dado um programa dividido em blocos básicos, a forma como estes se colocam relativamente uns aos outros pode influir no código produzido.

   Chama-se um *traço* a uma sequência de blocos básicos que possa ser executada no programa. Incluem-se os blocos terminados por um JUMP ou por um CJUMP indiferentemente.

   Iremos construir um conjunto de traços que cubra todo o programa (ie. que inclua todos os blocos básicos). Note-se que o conjunto final preserva o significado do programa pois este não depende da ordem relativa dos blocos, pelo que, se num traço correspondente a blocos que terminam com um JUMP não há dúvida relativamente ao que fazer, já num blocos que termine num CJUMP poderemos optar por construir um traço que siga por *qualquer* uma das duas saídas.

   A representação do programa ficará aqui como um conjunto de traços, cada um sendo uma sequência de blocos básicos.

3. **Observações finais**
   Um bom compilador procurará construir um conjunto de traços com o menor número de elementos possíveis.

   No nosso caso, e por uma questão de simplicidade de implementação, partiremos dum conjunto de traços em que os CJUMPs são tratados de forma indistinta dos JUMPs: lineariza-se simplesmente a lista de traços. Os critérios para permitir uma geração de código razoável deverão sobre os traços com CJUMPs:

   - Um CJUMP seguido pelo seu label `false` fica inalterado.

- Num CJUMP seguido pelo seu label `true`, troca-se os labels `true` e `false` e inverte-se o sentido da condição.
- Num CJUMP que não seja seguido por nenhum dos seus labels, constrói-se um artificial, para o `false`, em que se coloca um novo bloco básico com duas instruções: um LABEL e um JUMP.

3.15.3 Bibliografia

1. Capítulo 8 da referência principal.

3.16 Selecção de instruções

Independentemente da problemática da alocação de registos, a selecção de instruções aplica-se a uma árvore (canónica) de RI (ver secção 3.15 na página 45) por forma a associar grupos conexos de nós da RI a instruções da arquitectura alvo.

O resultado desta fase dum compilador é um programa *quase completo* para a arquitectura alvo, ficando por resolver a questão da alocação de registos que não será abordada nesta disciplina pois integra o programa da já referida disciplina **Complementos de Compilação**.

3.16.1 Algoritmos para selecção de instruções

O objectivo é obter uma cobertura total duma árvore de RI utilizando para tal únicamente sub-árvores correspondentes a instruções da arquitectura alvo. Trata-se dum problema semelhante aos de pesquisa em Inteligência Artificial, pelo que os algoritmos serão muito parecidos com os existentes nessa disciplina.

São apresentados alguns algoritmos de relativamente simples implementação (numa linguagem de programação imperativa, como o Java), nomeadamente:

- "Maximal munch", top-down sem retrocesso.
- Algoritmos de programação dinâmica, bottom-up.

Também é proposto aos estudantes o desenvolvimento duma implementação numa linguagem declarativa como o Prolog, por forma a verificarem a maior facilidade de expressão de soluções para situações como esta.

3.16.2 Máquinas CISC

Entende-se por "Máquina CISC" uma arquitectura que não "load-store", com relativamente poucos registos disponíveis e possívelmente com algumas instruções que efectuam muitas operações simultâneamente.

Estas características podem complicar o problema da geração de código optimo, pois a combinatória devida à diversidade de possíveis soluções cresce bastante.

São abordadas, do ponto de vista de geração de código, as arquitecturas VAX e Motorola 68000.

3.16.3 Selecção de instruções para TPL-03

A matéria descrita neste capítulo é aplicada à geração de código para a linguagem TPL-03.

3.16.4 Bibliografia

1. Capítulo 9 da referência principal.

4 Avaliação

A avaliação é feita essencialmente através da realização dum trabalho prático de grupo (máximo de 2 elementos), que vai evoluindo ao longo do semestre e é avaliado em 5 fases distintas.

Adicionalmente, é realizado um exame escrito que contará para 25% da nota, sendo os restantes 75% atribuídos ao trabalho prático.

A ênfase dada ao trabalho prático é, a meu ver, essencial para motivar os estudantes. O que tenho verificado nos anos anteriores é que esta abordagem atinge plenamente os seus objectivos, chegando mesmo a suscitar o entusiasmo dos estudantes mais empenhados.

4.1 Trabalhos práticos

Os trabalhos práticos têm os enunciados indicados na figura 6. Os alunos têm entre 2 e 3 semanas para efectuar cada trabalho.

---

1. Trabalho introdutório, no qual os estudantes se irão familiarizar com as ferramentas a utilizar nos restantes trabalhos. O objectivo é construir um reconhecedor para uma linguagem simples, tipo "calculadora evoluída", com variáveis, afectação e sequenciação de instruções.

2. Reconhecedor da linguagem TPL-03. Trata-se de construir um analisador lexical e sintáctico para TPL-03. O reconhecedor deverá ler do *standard input* um programa e escrever no *standard output* o texto `SIM` se este pertencer à linguagem e `NÃO` caso contrário.

3. Construção e visualização da árvore abstracta para a linguagem TPL-03. Este trabalho assenta e constitui um desenvolvimento sobre o resultado do trabalho 2. O formato de saída deverá ser aceite pelo programa de visualização `Outline`.

4. Geração de código SiM-03 para o compilador da linguagem TPL-03 já construído no trabalho 3. A linguagem pretendida é na realidade uma simplificação do TPL-03, no qual se omitem algumas características por forma a facilitar a implementação:

   - Tratamento de "strings".
   - "Arrays" com dimensões não constantes.

5. Continuação do trabalho 4, em que se propõe ao grupo implementar uma extensão à linguagem TPL-03 ou ao seu suporte em termos de compilador. No ano 2000/2001 as extensões propostas foram as indicadas na figura 7 (ver página 51).

Figura 6: Enunciados dos trabalhos práticos

1. **Instrução `case`**. Pretende-se uma instrução/expressão `case` que se assemelhe às homónimas noutras linguagens de programação imperativas, em termos sintacticos, semânticos e de desempenho. Proponha uma sintaxe concreta, a sintaxe abstracta e gere código SiM-03.

2. **Instrução `with`**. Pretende-se uma instrução/expressão `with` semelhante à existente no Pascal, i.e. que permita referir os campos duma variável dum tipo *record* sem referir a variável própriamente dita. Proponha uma sintaxe concreta, a sintaxe abstracta e gere código SiM-03.

3. ***Strings***. Defina um modelo de memória para usar strings na linguagem TPL-03 e implemente-o a partir do seu actual compilador de TPL-03 para SiM-03.

4. **Arrays dinâmicos**. Estenda o seu compilador de TPL-03 para SiM-03 por forma a este implementar arrays dinâmicos, i.e. arrays cuja dimensão possa incluir expressões não-constantes.

5. **Variáveis globais**. Especifique e implemente um mecanismo em que as variáveis globais (i.e. as que integram a que seria a "stack frame" de topo, a única de nível lexical 0) sejam atribuidas estáticamente, i.e. em endereços numa zona de memória fixa, fora da zona da pilha.

6. **Afectação paralela**. Modifique o seu compilador de TPL-03 para que este permita afectação paralela, por exemplo `(a, b, x[a]) := (b, a, x[a]+1)`. Proponha dias implementações: uma simples e outra claramente mais eficiente. Demonstre a geração de código.

7. **Passagem de parâmetros por referência**. Modifique o seu compilador de TPL-03 para que a passagem de parâmetros *possa* ser feita *por referência*. Defina uma sintaxe para a declaração destes parâmetros e implemente o que julgar necessário para atingir este objectivo.

8. **Parâmetros não posicionais e com valores por omissão**. Modifique o seu compilador de TPL-03 para que a declaração de funções possa incluir valores por omissão para os parâmetros. A linguagem também será estendida para permitir a activação de procedimentos usando os *nomes* dos parâmetros formais como se duma afectação se tratasse. Defina uma sintaxe contreta e implemente o que julgar necessário para atingir este objectivo.

Figura 7: Enunciados para o trabalho 5

4.2 Exame

No final da disciplina é efectuado um exame escrito, com duração típica de 4 horas, com consulta. Em anexo junto um exemplar do exame do ano anterior.

Enunciado da primeira chamada em 1999/00

---

1. **Análise Lexical e Sintactica**

   (a) Um analisador lexical pode ser implementado como um interpretador de DFAs, usando para tal duas tabelas: `próximo[estado,símbolo]` para representar as transições e `final[estado]` para designar quais os estados finais, sendo `final[i]` zero se o estado `i` não fôr aceitador e diferente de zero para designar o reconhecimento do token `final[i]`.

   Considerando o alfabeto $\Sigma = \{a, 0, +, (,), =\}$ designando duma forma abreviada o necessário para ler expressões aritméticas sobre variáveis (a designa letras, etc...), considerando que pretendemos reconhecer os tokens `ID`, `LIT`, `OP`, `LPAR`, `RPAR`, `EQ` (números respectivamente `0` a `5`):

      i. Escreva um conjunto de expressões regulares para reconhecer o pretendido.
      ii. Construa as tabelas `próximo` e `final` para o reconhecedor anterior.
      iii. Escreva em Java o esqueleto dum reconhecedor lexical que faça uso destas estruturas.

   (b) Considere o problema da análise sintactica de expressões aritméticas (com afectação, as 4 operações e funções unárias):

      i. Escreva uma gramática para esta linguagem.
      ii. Para a gramática anterior, esboce o código dum parser "predictivo" escrito em Java.
      iii. Indique o que faria para que o parser do ponto 1(b)ii faça interpretação semântica (i.e. avaliação).

2. **Análise Semântica**

   Considere que temos no Tiger uma instrução `with`, inspirada na homónima do Pascal.

   (a) Proponha uma sintaxe concreta e uma sintaxe abstracta para esta instrução.

   (b) Indique as características desejáveis para uma hierarquia de classes `Symbol.Table` para suportar eficazmente esta construção. Especifique a interface e esboce o código.

   (c) Depois de efectuada a análise de nomes e de tipos, qual é a influência da inclusão de instruções `with` no programa? Comente.

   (d) Quais são os passos a efectuar na análise de nomes e de tipos para a inspecção da instrução `with`?

3. **Geração de Código**

   Considere a produção de código para a arquitectura SiM. Suponha que pretendemos representar os arrays no stack. Nesta questão omita a existência de records.

   (a) Que restrições deveremos impôr aos tipos de funções e outros blocos para manter a invariante do valor de retorno ser um escalar (i.e. representável numa só palavra)? Comente.

   (b) Indique o código do prólogo e epílogo dos blocos onde estão declaradas variáveis dum tipo *array*.

   (c) Escreva o código SiM para aceder a `xpto[e]` em leitura, considerando que `xpto` é uma array de `int` local ao bloco em que ocorre o acesso.

---

5 Bibliografia e Software de apoio

A bibliografia recomendada para esta disciplina é a seguinte:

1. Andrew A. Appel, *Modern Compiler Construction in Java*. Cambridge University Press, 1998. ISBN: 0-521-58388-8. **Referência Principal**.

   Este livro é usado como guia para a maior parte das aulas teóricas assim como fonte de exercícios para as aulas práticas.

2. Alfred Aho, Ravi Sethi, Jeffrey Ullmann, *Compilers: Principles, Techniques and Tools*. Addison-Wesley 1986. ISBN: 0-201-10194-7.

   Este livro (conhecido como o "Dragon Book") constitui leitura indispensável para os estudantes que pretendam aprofundar mais os seus conhecimentos.

3. James Gosling, Bill Joy, Guy Steele, *The Java Language Specification*. Addison-Wesley 1996. ISBN: 0-201-63451-1. Disponível livremente em formato PDF.

   A referência "On-line" para a linguagem Java e a sua livraria de classes base. Note-se que este documento omite as classes de interface utilizador, que são dispensáveis no âmbito desta disciplina.

O software utilizado para desenvolvimento dos trabalhos práticos consiste nos seguintes sistemas, todos eles imediatamente disponíveis aos estudantes:

- Ferramentas de programação em Java JDK versão 1.1.8 ou versão J2SDK 1.3.0 da Sun. Ambas disponíveis directamente no ambiente de programação do Linux Debian.
- Gerador de "parsers" para Java: CUP (`http://www.cs.princeton.edu/~appel/modern/java/CUP/`) e BYACC/j (`http://troi.lincom-asg.com/~rjamison/byacc/`).
- Gerador de analisadores lexicais JLex (`http://www.cs.princeton.edu/~appel/modern/java/JLex/`) e JFlex (`http://www.jflex.de/`).
- Visualizador de estruturas arborescentes `Outline`, da minha autoria e adequado para a visualização de árvores abstractas por intermédio duma representação textual simples.
- Ferramentas adicionais, das quais destaco:
  - Gerador de "parsers" para Java: ANTLR (`http://www.antlr.org/`).
  - Blackdown Linux JDK. É uma implementação do Java Development Kit adaptada ao sistema Linux.
  - IBM JDK. Trata-se duma implementação da máquina Java mais eficiente do que a do JDK standard, pelo que poderá ter interesse para quem estiver disposto a arriscar eventuais pequenas incompatibilidades.
  - GNU Prolog, que pode ser utilizado para implementar partes do compilador de forma mais cómoda.

6 Sumários das aulas teóricas

Estes irão simplesmente referir as partes do programa anteriormente descrito na secção 3 pelo que terão simplesmente como indicação o número da aula e o índice da matéria discutida. O plano da matéria para cada aula está apresentado no quadro 7.

Aula	**Matéria**	
1	3.1	Introdução
2	3.2	Análise Lexical
3	3.3	Análise sintáctica
4	3.4	A linguagem TPL-03
5	3.8	Sintaxe abstracta
6	3.9	Dicionários (tabelas de símbolos)
7	3.10	Análise semântica (Nomes e Tipos)
8	3.11	Registos de activação
9	3.12	A máquina de pilha SiM-03
10	3.13	Geração de código para a máquina SiM-03
11	3.13	Geração de código para a máquina SiM-03 (complementos)
12	3.14	Geração de código intermédio
13	3.15	Blocos básicos e traços
14	3.16	Selecção de instruções

Quadro 7: Programação das aulas teóricas

Considerando que cada aula teórica tem a duração de 2 horas e que os estudantes têm conhecimento antecipado de qual a matéria a apresentar, este tempo será suficiente para que possam ser esclarecidas dúvidas sobre os pontos mais difíceis ou ambíguos da matéria.