



UNIVERSIDADE TÉCNICA DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

COMPILADORES

Pedro Reis dos Santos

Departamento de Engenharia Informática

14 de fevereiro de 2012

Compiladores₂₀₁₂, Apresentação

0-1

Objetivos

- Compreender e definir informalmente uma linguagem livre de contexto.
- Desenvolver a capacidade de resolver problemas que obrigam à tradução entre linguagens ou notações, utilizando os métodos de análise (*front-end*) e síntese (*back-end*) mais usados pelos compiladores.
- Conhecimento dos processos, ferramentas, algoritmos e estruturas de dados utilizados na manipulação de linguagens, sejam de programação ou não.
- Experimentação de produção de software na realização de um compilador para uma linguagem de programação simples.
- A disciplina faz uso de grande número de conceitos lecionados anteriormente: fundamentos de programação, sistemas digitais, teoria da computação, **arquitetura de computadores, algoritmos e estruturas de dados, sistemas operativos, programação por objetos.**

Compiladores₂₀₁₂, Apresentação

0-2

Resultados

- Compreender as linguagens, de programação ou não, nomeadamente a sua estrutura (sintaxe), semântica e limitações. [FP, PO]
- Assimilar as técnicas de análise de linguagens, ou qualquer formato de dados com estrutura predefinida. [TC, AED]
- Tomar conhecimento das estruturas de dados e técnicas envolvidas na geração de código máquina. [SD, AC, SO]

Compiladores₂₀₁₂, Apresentação

0-3

Gramáticas

- Uma linguagem é a expressão de pensamentos por frases e palavras.
- Uma gramática de uma linguagem é um conjunto de regras que permitem definir a linguagem.
- Pode haver mais de uma gramática para a mesma linguagem: gramáticas equivalentes.
- Hierarquia das gramáticas propostas por Noam Chomsky em 1956:
 - tipo 0:** Sem restrições (português, por exemplo)
 - tipo 1:** Sensíveis ao contexto
 - tipo 2:** Livres de contexto
 - tipo 3:** Regulares

Compiladores

Compilador: programa que lê um programa descrito numa linguagem (fonte) e o traduz para outra linguagem (destino), reportando os erros quando eles ocorrem.

Processo de compilação: designa o conjunto de tarefas que o compilador deve realizar para poder gerar uma linguagem a partir de outra.

Fases do processo de compilação: análise (lexical, sintática e semântica) e síntese (geração e otimização de código).

Fase de análise: quebra o código fonte nas suas partes, e cria uma representação intermédia do programa (tabela de símbolos e árvore de parsing).

Fase de síntese: constroi o programa-destino a partir de uma representação intermédia (árvore, DAG ou dirigido pela sintaxe).

Ambiente de desenvolvimento

- O ambiente de desenvolvimento utilizado na disciplina de Compiladores utiliza o sistema operativo **linux** em processadores **i386** (e derivados **Pentium ...**).
- O compilador é desenvolvido em **C++**.
- Biblioteca de runtime é código **i386**, podendo ser escrita em **C**, **C++** ou a linguagem de projeto se o compilador já estiver operacional.
- A escolha da linguagem a usar deve ter em conta a experiência de programação em cada uma das linguagens e as características específicas de cada linguagem.
- Existe suporte para desenvolvimento em Windows mas a avaliação final do projeto é efetuada em **linux**.

Ferramentas de desenvolvimento

- Básicas: editores de texto (vi/emacs), linguagem de comando (sh/csh), compilador de C++ (g++) ou C (gcc).
- Depurador de código ou *debugger*: gdb, ddd.
- Controlo de versões: cvs.
- Controlo de configurações: gmake.
- Assembler (geração de código relocatável): nasm.
- Edição de ficheiros objeto (código relocatável): ld, ar.
- Manipulação de ficheiros objeto: nm, size, strip, strings, od, objdump, ...
- Monitores (*debug* e otimização): strace, gprof.

Qualidade do código

- Os nomes das funções e variáveis devem ser autoexplicativos.
“Quando o código e respetivos comentários discordam, provavelmente estão ambos errados” - *Norm Schreyer*
- A legibilidade é mais importante que a eficiência, número de linhas, etc.

```
int i;main(){for(;i["<i;++i){--i;}"];read('-'-'-',i+++hell\
o, world!\n",'/'/'/'))};read(j,i,p){write(j/p+p,i---j,i/i);}
Dishonorable mention, Obfuscated C Code Contest, 1984. Author requested anonymity.
```
- As rotinas de biblioteca utilizadas devem existir nas normas: ANSI C (ANSI X3.159-1989), ISO C99 (ISO/IEC 9899:1999) ou POSIX (POSIX 1003.1).
“O C combina o poder do *assembly* com a portabilidade do *assembly*” - *Bill Thacker* alegadamente
- O ambiente de trabalho é **linux-elf-i386**, logo as linhas não terminam em CR-LF nem os ficheiros em ^Z sem CR, por exemplo.

Programa

- Ambiente de desenvolvimento: ferramentas.
- Linguagens regulares e análise lexical.
- Linguagens livres de contexto e gramáticas atributivas.
- Análise sintática: descendente e ascendente.
- Análise semântica: estática e dinâmica.
- Geração de código: intermédio e final.
- Seleção e escalonamento de instruções.
- Reserva de registos.
- Otimização de código: análise de fluxo, otimização local e global.

Avaliação

- Informação: **página da disciplina no fénix**
não se tiram dúvidas de matéria por mail.
- Avaliação: 40% projeto e 60% exame.
- Teórica (min. 9,5):
2 testes (24 março e 4 junho) + repescagem (29 de junho)
- Projeto (min. 9,5): em grupos de 2 elementos, com teste prático individual.
2 entregas no fénix (16 abril e 21 maio) e teste prático (22 maio).
 $\text{proj} = \min(\text{cod}; \text{pra}) + \sqrt{\text{diff}(\text{cod}; \text{pra})}$ onde $\text{cod} = (\text{p1} + \text{p2} + 2 \cdot \text{docente}) / 4$.
acesso ao teste prático tem nota mínima de 8 valores.
- Aulas práticas: realização pelos alunos de exercícios de exame, com acompanhamento do docente.
Época especial (40% proj e 60% exame) tem projeto específico.

Projeto

- Manual de Referência descreve a linguagem.
- Enunciado do Projeto descreve o conteúdo de cada entrega e o processo para o produzir.
- Entregas feitas no **fénix** até às 12h (hora do servidor):
intermédia: lex e yacc, até 16 de abril.
final: semântica e geração de código, até 21 de maio.
- Testes práticos a 22 de maio.
- Repetentes são obrigados a repetir toda a avaliação, incluindo o projeto.

Bibliografia

- “Compiladores: da teoria à prática”, Pedro Reis dos Santos, IST 2006
- “Compiladores: ferramentas de desenvolvimento”, Pedro Reis dos Santos, 4^a edição, IST 2006
- “Compilers: Principles, Techniques, and Tools”, Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Addison-Wesley, 1986, 0-201-10088-1
- “Processadores de linguagens: da concepção à implementação”, Rui Gustavo Crespo, IST press, 2^a edição, 2001, 972-8469-18-7
- “lex & yacc”, John R. Levine, Tony Mason, Doug Brown, O’Reilly & Associates, 2^a edição, 1992, 1-56592-000-7

Introdução

Compilador: programa que lê um programa descrito numa linguagem (fonte) e o traduz para outra linguagem (destino), reportando os erros quando eles ocorrem.

Processo de compilação: designa o conjunto de tarefas que o compilador deve realizar para poder gerar uma linguagem a partir de outra.

Fases do processo de compilação: análise (lexical, sintática e semântica) e síntese (geração e otimização de código).

Fase de análise: quebra o código fonte nas suas partes, e cria uma representação intermédia do programa.

Fase de síntese: constrói o programa-destino a partir de uma representação intermédia.

Compilador

Verificador: verifica se o formato de entrada obedece a um determinado formato. A resposta é, em geral, **sim** ou **não**.

Exemplos: *syntax highlighting* em editores de texto, verificação de normas.

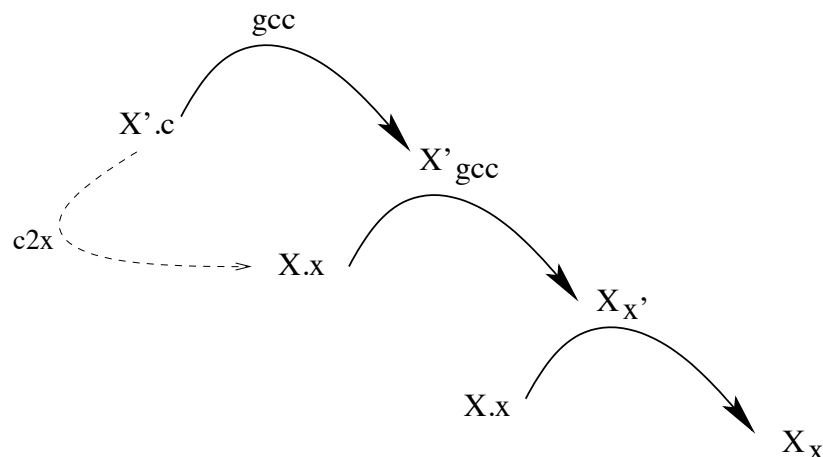
Interpretador: executa o código, depois de o verificar, produzindo a partir de uma entrada de dados (distinta da entrada de código) um conjunto de resultados. O processo permite, segundo o código, executar cálculos, atualizar bases de dados, construir estruturas de dados, ...

Exemplos: emuladores, interpretadores e simuladores.

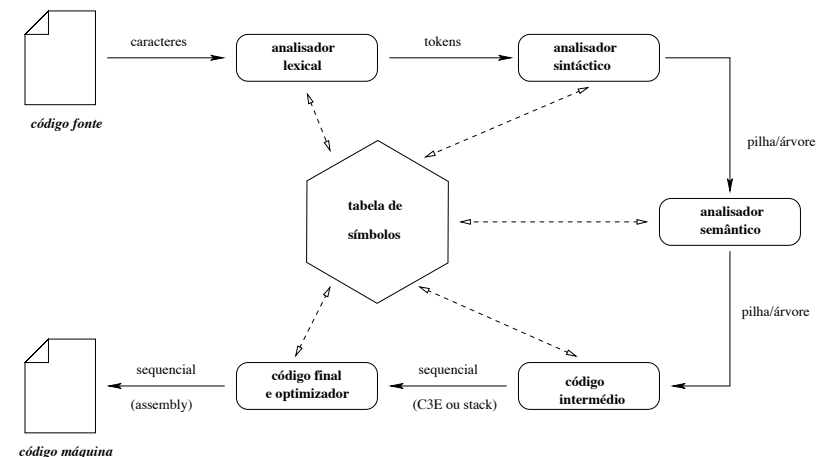
Tradutor: traduz uma determinada linguagem de entrada numa linguagem de saída. Quer a linguagem de entrada como a de saída podem ser de nível máquina, *assembly* ou de alto nível.

Exemplos: compiladores, *beauty printers*.

Geração de um compilador



Processo de compilação



Análise lexical

- Agrupa os caracteres em símbolos e verifica se pertencem à linguagem: palavras reservadas, identificadores, literais, operadores, delimitadores, comentários e separadores.

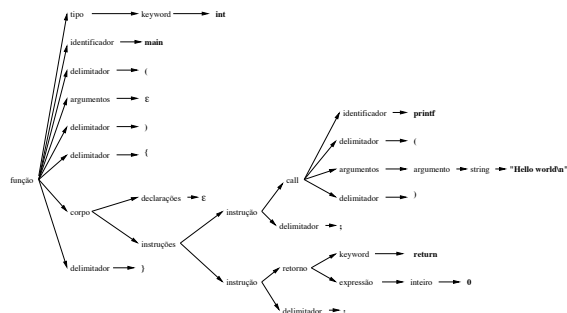
```
int main ( )
{
    printf ( "Hello world\n" ) ;
    return 0 ;
}
```

- Erro lexical (ortográfico): "A **bolaxa** comeu uma João"

NADA PERCEBER SEM ISTO TENTAR LER A ESTOU QUE TEMPO ALGUM JÁ HÁ

Análise sintática

- Verifica se as sequências de símbolos são válidas na linguagem.
- O emparelhamento das regras resulta numa árvore sintática:



- Erro sintático: "A bolacha comeu **uma** **João**"

Análise semântica



Análise semântica

- Verifica se as entidades podem desempenhar as funções pretendidas e determina como as devem desempenhar.
- manipulação de identificadores: visibilidade, alcance, tabelas de símbolos
- Tipificação (verificação de tipos): equivalência, subtipificação, polimorfismo.
- Árvores de ativação (chamadas a procedimentos): gestão de memória, registos de ativação (stack frames), ligação de nomes (binding).
- Por exemplo, em C pode incluir: `break` fora de ciclos, `return` em funções `void`, número ou tipo dos argumentos das funções incorreto, retirar endereço de constantes, atribuições a constantes, utilização de variáveis não declaradas, redefinição de funções ou variáveis, *etc.*
- Erro semântico: “A bolacha comeu o João”

Tratamento e recuperação de erros

- O objetivo é a correção: não interessa quanto tempo se gasta para obter uma resposta a um erro!
- Todos os erros no texto são sintáticos, mas muitos erros não podem ser tratados por linguagens independentes do contexto.
- Erros não detectáveis, ou dificilmente detectáveis, com a tecnologia atual são processados pela análise semântica estática.
- Tipos de respostas a um erro: inaceitáveis (incorretas ou de pouca utilidade) e aceitáveis.
- Tratamento de erros através da inserção na gramática de regras específicas.

Interpretação

- Processamento das instruções de um programa por uma aplicação e não pelo processador da máquina.
- A cada primitiva da linguagem corresponde um pedaço de código no interpretador.
- Interpretação pode ser:
 - direta, linha a linha, do texto do programa.
 - dirigida pela sintaxe (em gramáticas simples).
 - dirigida pela árvore sintática (100x a 1000x mais lento).
 - execução de código para máquina virtual ou *threading* (10x a 100x).
- Código intermédio para processadores ideais, com zero ou infinitos registos, pode ser interpretado ou usado para gerar código final.

Geração de código

- Produção de código para um processador específico, tendo em conta as suas características particulares.
- Tipos de processadores podem ser:
 - Stack machines: B5000, HP-3000 ou máquinas virtuais **Java**, e **.net**.
 - Accumulator machines: PDP-9, M6809, 68HC11.
 - Load-store machines (RISC): ARM, RS/6000, MIPS-R, HP-PA, SPARC.
 - Memory machines (CISC):
 - * Register-memory architecture: i80x86, IBM-RT, M68k, IBM360.
 - * Memory-memory architecture: VAX, PDP-11.
- Compilação JIT a partir de um formato intermédio ou compilação *object-code* a partir de código final.

Código i386 assembly

- Código em assembly (formato **nasm**) do hello world:

```
segment .rodata
    $str    db 'hello world',10,0
segment .text
extern printf
global main
main:
    push    dword $str
    call    printf
    add     esp, 4
    mov     eax, 0
    ret
```

Código i386 objeto

- Resultado no ficheiro objeto (binário):

```
Disassembly of section .rodata:   (13 bytes)
00000000 <str>:
0:  68 65 6c 6c 6f 20 77 6f 72 6c 64 0a 00
Disassembly of section .text:   (22 bytes)
00000000 <main>:
0:  68 00 00 00 00          push    $0x0 <str>
5:  e8 fc ff ff ff         call    6 <printf>
a:  81 c4 04 00 00 00      add     $0x4,%esp
10: b8 00 00 00 00         mov     $0x0,%eax
15:  c3                     ret
```

Otimização de código

- preservar o significado do programa original (controlabilidade e observabilidade).
- em média, deve melhorar visivelmente o tempo de execução, uso da memória ou espaço ocupado pelo programa.
- deve ter uma boa relação custo benefício.
- processo: análise do fluxo de controlo, análise do fluxo de dados, transformações, ...
- otimizações: alto nível (programador), local, peephole, global.

Código i386 otimizado

- Resultado no ficheiro objeto otimizado para espaço:

```
Disassembly of section .rodata:   (13 bytes)
00000000 <str>:
0:  68 65 6c 6c 6f 20 77 6f 72 6c 64 0a 00
Disassembly of section .text:   (14 bytes)
00000000 <main>:
0:  68 00 00 00 00          push    $0x0 <str>
5:  e8 fc ff ff ff         call    6 <printf>
a:  58                     pop     %eax
b:  31 c0                 xor     %eax,%eax
d:  c3                     ret
```

- Clocks (80C386): **push imm** (3), **pop reg** (5), **call imm** (7), **ret** (11), **add imm** (3), **xor reg** (2).
- Ocupa menos 8 bytes mas gasta mais 2 ciclos de relógio (**mov** vs. **pop**).

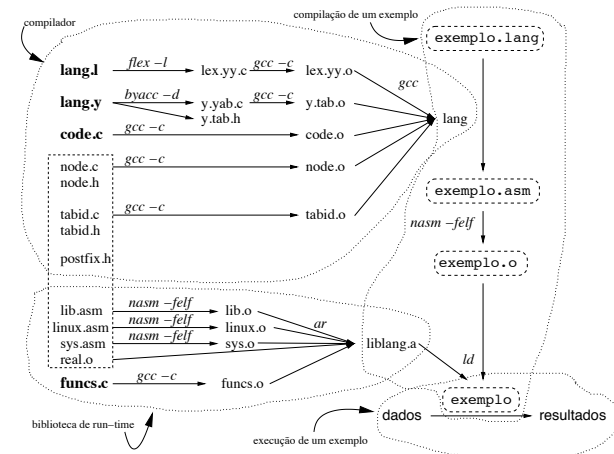
Desenvolvimento de compiladores de compiladores

— Compiladores de compiladores são ferramentas em que são elas que geram o código do compilador (C/C++).

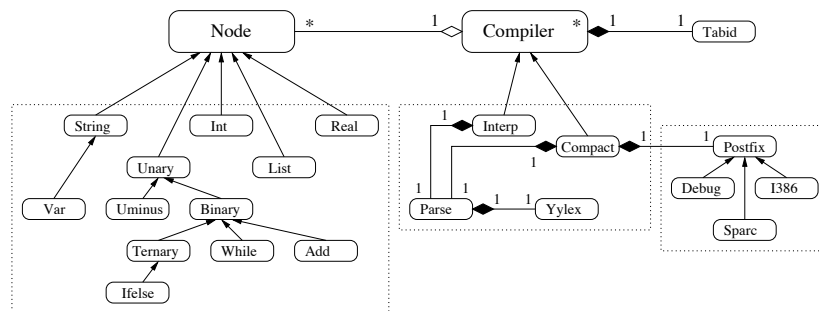
O programador escreve uma descrição de mais alto nível.

- Analisadores léxicos: lex, Jlex, **flex**, ...
- Analisadores sintáticos: yacc, Jcup, bison, **byacc**, precc, antlr, ...
- Avaliadores de atributos: ox, rie, ...
- Selecção de instruções: iburg, **pburg**, lburg, ...

Projeto de Compiladores em C



Compilador Orientado por Objetos



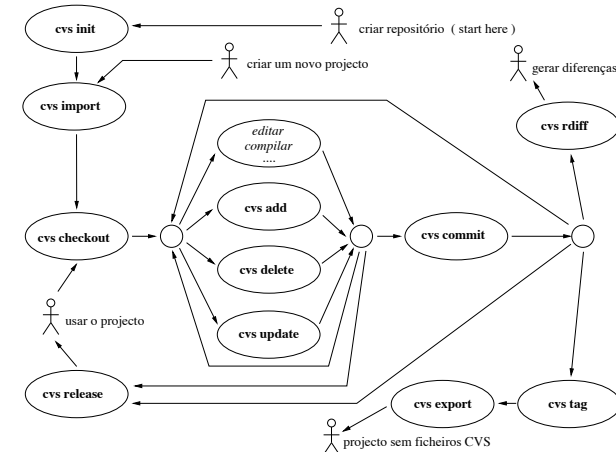
Controlo de versões

- Mantém as diversas versões de um ficheiro, numa estrutura em árvore, guardando apenas as diferenças entre versões consecutivas.
- Mantém uma história dessas alterações, ligando-as ao autor, data e respetiva justificação.
- Permite linhas de desenvolvimento paralelas, podendo ser fundidas em determinada altura.
- Mais de um utilizador pode modificar a mesma versão: deteta conflitos e dispõe de mecanismos para os eliminar (automáticos e manuais).
- Permite a identificação automática dos ficheiros: data, versão, autor, ...
- Cada utilizador tem uma área de trabalho (*sand-box*) a partir da qual faz *check-out* e *check-in*, **de e para** o repositório (ou base de dados).

Sistemas de controlo de versões

- Entidade manipulada:
 - Ficheiro: cada ficheiro é manipulado individualmente pelo utilizador (RCS).
 - Projeto: o utilizador manipula projetos completos (CVS).
- Controlo de acesso:
 - Pessimista: é necessário reservar o acesso para escrita no *check-out* (RCS).
 - Otimista: todas as cópias são modificáveis, os conflitos são resolvidos no *check-in* (CVS).
- Interação com o sistema de controlo de configurações:
 - Direto: o controlo de configurações gere as versões (RCS).
 - Inverso: o controlo de versões gere as configurações (CVS).

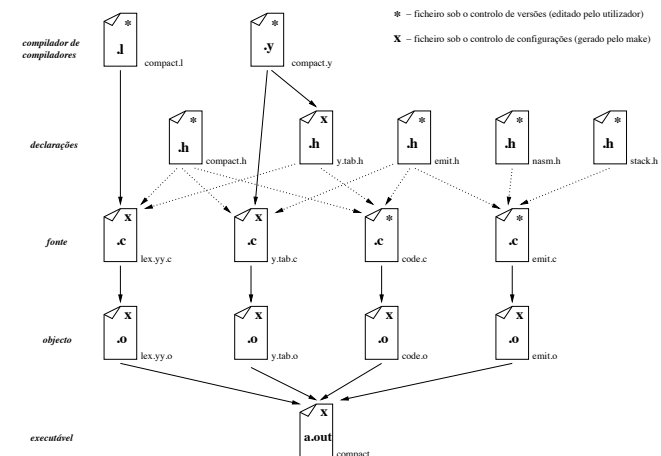
CVS



Controlo de configurações

- Controla e automatiza a produção de ficheiros geráveis mecanicamente: baseado em regras simples.
- Regra: se **b** é gerado a partir de **a**, então quando **a** é modificado **b** tem de ser gerado.
- O controlo é efetuado por datas de modificação (mtime): as datas mostradas no **ls -l**.
- Uma configuração é uma **receita** (Makefile) onde se indica quem depende de quem, e como fazer quando as dependências falham.
- O processo de desenvolvimento, uma vez construída a configuração, resume-se a: pensar → editar → **make** → testar.

Grafo de dependências



Regras

- Componentes de uma regra:

objetivo : o que se pretende fazer (make), pode não ser um ficheiro.

dependente : de quem depende o objetivo, devendo os comandos ser executados se a sua data for posterior ao objetivo.

operação : sequência de comandos a ser executados quando a regra é ativada.

- Sintaxe da regra: **objetivo: [dependentes... [; operações...]]**
ou numa notação mais legível:

```
objetivo : dependente-1 dependente-2 ... dependente-N
           operação-1
           operação-2
           ...
           operação-N
```

Definições

- Uma definição é uma diretiva de substituição, não é uma atribuição segundo o modelo de *von-Neumann*: **ID = valores**
- A substituição é efetuada com **\$(ID)** ou **\${ID}**, exceto se for uma única letra **\$ID**.
- Variáveis de ambiente do *shell* são acessíveis dentro da *Makefile*.
- As definições dinâmicas são relativas à regra onde se incluem: **\$<**, **\$?**, **\$@**, **\$***, **\$^**, **\$+**, *etc.*
- A substituição de uma definição segue a ordem: linha de comando, makefile, variáveis de ambiente, predefinições.
- Definições com padrões: **\$(var:b1%s1=b2%s2)** substitui **b1%s1** por **b2%s2** em **var**, onde **'%'** representa zero ou mais caracteres.

Regras implícitas

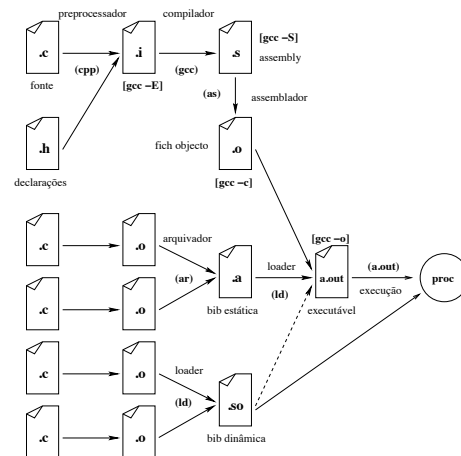
- Um regra implícita estabelece uma dependência entre conjuntos de ficheiros, não entre ficheiros individuais.
- Regras implícitas por extensão:
.cpt.asm: ; compact \$< > \$@
- Regras implícitas por padrões:
% : %.asm ; nasm -felf \$< ; ld \$*.asm -o \$@
- Ordem de emparelhamento das regras: explícitas, implícitas na *Makefile*, implícitas predefinidas.
- Em caso de conflito no emparelhamento, ou quando se introduzem novos sufixos, **make** usa a ordem de sufixos no pseudo-objetivo **.SUFFIXES**:
.SUFFIXES: .cpt .c .p .f .asm .s .o

Makefile

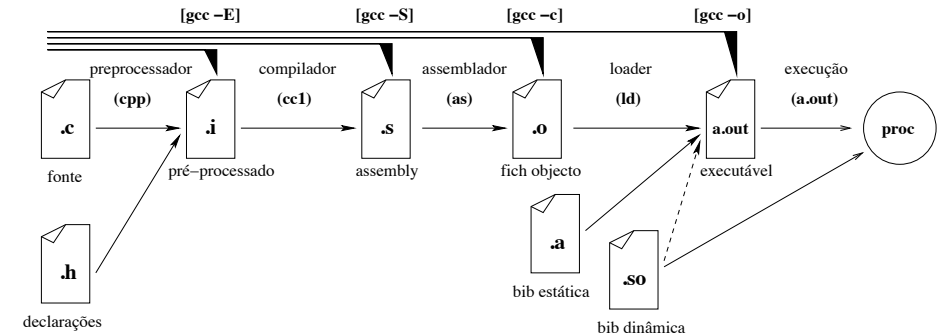
- Linhas podem ser continuadas se terminarem com **'\'**.
- Comentários iniciam-se com **'#'** e terminam no fim da linha.
- regras sem dependentes executam sempre que ativadas.
- O objetivo principal é a regra chamada **all**, ou a primeira regra não implícita do ficheiro, caso não exista a regra **all**.

```
YFLAGS=-d
LEX=flex -l
OBJ=gram.o scan.o code.o emit.o
compiler: $(OBJ)
           $(LINK.c) -o $@ $^
scan.o: gram.y
$(OBJ): defs.h
```

Compilador de C



gcc



Depuração de código

- *debugger* (adb/gdb): execução controlada, passo a passo, permitindo inspecionar as instruções máquina executadas, bem como ler e modificar valores em posições de memória (dados).
- *debugger* simbólico (gdb): *debugger* onde as instruções são as linhas do ficheiro fonte e as variáveis são referidas por nome.
- Programas monitores: programas que permitem observar o comportamento de outros sem interferir (supostamente) no seu funcionamento.
 - trace (strace): intercetor de chamadas ao sistema.
 - time: tempo de execução e recursos do sistema operativo.
 - prof (gprof): numero de chamadas e tempo gasto por rotina do programa.

Objetos e executáveis

- Um ficheiro objeto relocatável é o resultado de uma compilação onde os endereços dos símbolos são relativos à origem do segmento.
- Um editor de ligações (*linker*, designado **ld** em UNIX) permite combinar objetos relocatáveis resolvendo os símbolos comuns.
- Um objeto relocatável pode ser criado através de ligação de diversos outros objetos relocatáveis, resolvendo os símbolos comuns. [*ld -r*]
- Um executável é um ficheiro, com um ponto de entrada, sem símbolos indefinidos e não relocatável (endereços absolutos).
- Um executável com ligações estáticas contém o código de todas as funções que necessita. [*ld -Bstatic*]

Bibliotecas

- Uma biblioteca (estática) é uma coleção de ficheiro objetos relocatáveis, mantidos independentemente. [*ar t libZ.a*]
- Uma biblioteca dinâmica é um objeto não relocável com uma tabela de símbolos dinâmica. [*ld -E -shared*]
- Um executável que use bibliotecas dinâmicas não as inclui, apenas as refere. No início da execução as bibliotecas são carregadas logo após o executável.
- Uma biblioteca dinâmica pode referir símbolos globais do executável em que se integra. [*ld -E -Bsymbolic -shared*]
- Um executável pode ter a capacidade de carregar bibliotecas dinâmicas durante a execução, se tiver uma tabela de símbolos dinâmica. [*ld -E*] (ou [*gcc -rdynamic -ldl*])

Linguagens regulares

alfabeto: conjunto finito de símbolos, designados por palavras.

frase: sequência finita de palavras de um dado alfabeto.

sintaxe: conjunto de regras que delimita o subconjunto de frases constituintes de uma linguagem.

linguagem regular: é descrita por uma expressão regular.

expressão regular: define uma linguagem com base nas regras:

- \emptyset é uma expressão regular que define a linguagem vazia.
- ε é uma expressão regular que define a linguagem que consiste na frase nula $\{\varepsilon\}$.
- um símbolo do alfabeto é um expressão regular que define a linguagem formada pela frase constituída por esse símbolo.
- um conjunto de operadores sobre expressões regulares.

Operadores das expressões regulares

Uma expressão regular pode ser formada pelos seguintes operadores, considerando duas expressões regulares p, q :

escolha: designado por $p|q$ é comutativo e associativo, representando a união das expressões regulares originais.

concatenação: designado por $p q$ é associativo e distributivo em relação à escolha e mais prioritário que a escolha. ε é elemento neutro na concatenação.

fecho de Kleene: designado por p^* é idempotente ($p^* p^* = p^*$) e mais prioritário que a concatenação e representa o conjunto de frases constituídas por zero ou mais repetições das frases de p . Além disso $p^* = (p|\varepsilon)^*$.

parênteses: permite alterar a prioridade dos operadores anteriores.

Extensões aos operadores das expressões regulares

opção: designado por $p?$ designa zero ou uma ocorrências de p .
É equivalente a $p|\varepsilon$.

fecho transitivo: designado por p^+ designa uma ou mais ocorrências de p .
 p^+ é equivalente a $p p^*$, e p^* é equivalente a $p^+ |\varepsilon$.

parênteses retos: designado por $[pq]$ ou $[p - q]$ designam $p|q$ e os símbolos de ordem entre p e q (inclusivé), respetivamente.

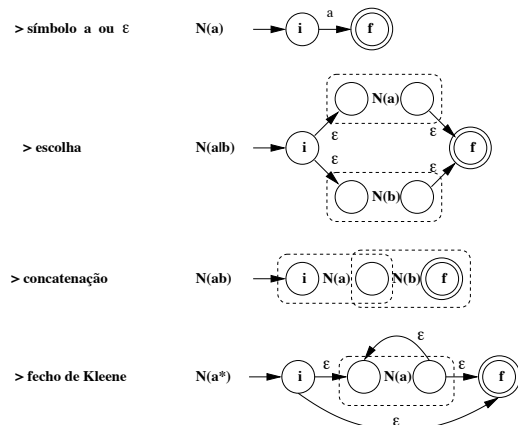
Diagrama de transição

- O reconhecimento de linguagens regulares, frases geradas por expressões regulares, pode ser modelado por autômatos finitos.
- Um autômato finito pode ser representado por um **diagrama de transição**:
 - um conjunto de estados, representados por círculos.
 - transições entre estados representadas por setas etiquetadas por símbolos de entrada.
 - um ou mais estados finais, representados por círculos duplos concêntricos.
 - um estado inicial, indicado por uma seta sem origem em outro estado.

Autômato finito não determinista: AFND

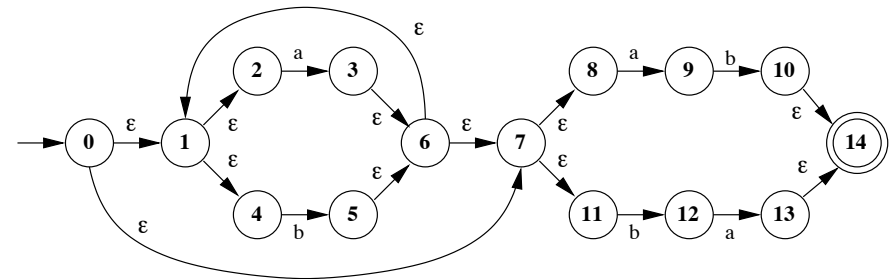
- Num autômato finito não determinista onde podem existir transições sem símbolo de entrada, sendo etiquetadas por ϵ .
- Num AFND pode existir mais de uma sequência de transições que permita atingir um estado final a partir do estado inicial.
- A procura de solução num AFND não é determinista, pelo que pode ser necessário recuar algumas transições para procurar alternativas (*backtracking*).
- A construção de um AFND a partir de uma expressão regular é algorítmica: algoritmo de Thompson.

Algoritmo de Thompson



Exemplo de AFND

Autômato finito não determinista construído segundo o algoritmo de Thompson para a expressão regular $(a|b) * (ab|ba)$:



Autômato finito determinista: AFD

- Num autômato finito determinista não podem existir transições vazias (etiquetadas por ε).
- Em cada estado existe, no máximo, uma só transição etiquetada com determinado símbolo de entrada.
- Num AFD só existe uma sequência de transições que permite atingir um estado final a partir do estado inicial.
- A construção de um AFD a partir de um AFND é algoritmica: construção de sub-conjuntos ($fecho - \varepsilon$ para cada símbolo de entrada em todos os estados).
- O processamento de um AFD é baseado numa tabela, sendo simples e eficiente, em termos de tempo de execução e espaço ocupado.

Construção de um AFD a partir de um AFND

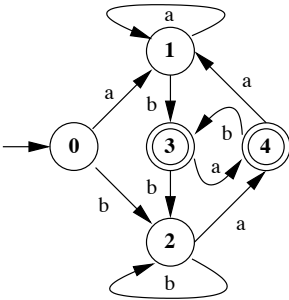
- $fecho - \varepsilon(i_0)$ a partir do estado inicial do AFND i_0 determinar o conjunto de todos os estados que podem ser atingidos apenas através de transições vazias ε ($fecho - \varepsilon(i_0)$), este é o estado inicial do AFD I_0 . (notar que $i_0 \in I_0$)
- $move(I_0, a)$ para cada símbolo de entrada, calcular o conjunto dos estados do AFND que podem ser atingidos com uma transição etiquetada com esse símbolo.
- $fecho - \varepsilon(move(I_0, a))$ para cada conjunto dos estados identificados na alínea anterior calcular o respetivo $fecho - \varepsilon$ até não haver novos estados.

Exemplo de construção de um AFD

estado	entrada	move	$fecho - \varepsilon \backslash move$	novo estado
		0	1, 2, 4, 7, 8, 11	I_0
I_0	a	3, 9	6, 1, 2, 4, 7, 8, 11	I_1
I_0	b	5, 12	6, 1, 2, 4, 7, 8, 11	I_2
I_1	a	3, 9	...	(I_1)
I_1	b	5, 10, 12	14 , 6, 1, 2, 4, 7, 8, 11	I_3
I_2	a	3, 9, 13	14 , 6, 1, 2, 4, 7, 8, 11	I_4
I_2	b	5, 12	...	I_2
I_3	a	3, 9, 13	...	(I_4)
I_3	b	5, 12	...	(I_2)
I_4	a	3, 9	...	(I_1)
I_4	b	5, 10, 12	...	(I_3)

Tabela de transição

est	a	b
I_0	I_1	I_2
I_1	I_1	I_3
I_2	I_4	I_2
I_3	I_4	I_2
I_4	I_1	I_3



Minimizar os estados de um AFD

- o número de estados obtidos pelo algoritmo de conversão de AFND para AFD não é o mínimo.
- algoritmo de minimização do número de estados de um DFA:
 - construir uma partição inicial com 2 grupos: um grupo contém todos os estados finais e o outro grupo os restantes estados.
 - construir uma nova partição **agrupando** os estados de cada grupo de partida que, para cada símbolo de entrada, têm transições para estados de um mesmo grupo de partida, qualquer que seja o símbolo de entrada.
 - repetir a alínea anterior até que não seja possível criar mais grupos.

Gramática regular

- do autômato finito determinista minimizado pode-se obter diretamente uma gramática regular.
- a gramática regular contém tantos símbolos não terminais quantos os estados do autômato.
- cada transição $\delta(q_i, a) = q_j$ produz numa regra $S_i \rightarrow aS_j$
- cada estado final q_f produz uma regra $S_f \rightarrow \varepsilon$.
- a tabela de análise e gramática regular de $(a|b)^*abb$

estado	a	b
0	1	0
1	1	3
3	1	4
4	1	0

$S_0 \rightarrow a S_1 \mid b S_0$
 $S_1 \rightarrow a S_1 \mid b S_3$
 $S_3 \rightarrow a S_1 \mid b S_4$
 $S_4 \rightarrow a S_1 \mid b S_0 \mid \varepsilon$

Analísadores *hardcoded*

- abordagem baseada em saltos (*gotos*), do estado 4:

```
state4:
    in = *input++;
    if (in == 'a') goto state1;
    if (in == 'b') goto state0;
    if (in == 0) return 1; /* só para estados finais */
    goto error; /* termina cada estado */
```

- abordagem baseada em funções:

```
static int state4() {
    register char in = *input++;
    if (in == 'a') return s1();
    if (in == 'b') return s0();
    return 1; /* estados não finais retornam 0 */
}
```

Analísador lexical

- O estado inicial inclui transições vazias para os estados iniciais das expressões regulares.
- Na busca da maior expressão possível, como irão existir diversos estados terminais, deve-se avançar até um erro ou fim dos dados e procurar o último estado terminal aceite.
- A análise recomeça no estado inicial global, no carácter seguinte ao que conduziu ao estado final aceite.
- Na minimização, a partição inicial coloca em grupos diferentes os estados terminais de cada uma das expressões regulares.

Tempo de computação e ocupação de memória

- O tempo de computação e a ocupação de memória dos autómatos finitos depende do comprimento da expressão regular $|r|$ e do comprimento da cadeia a reconhecer $|x|$.
- Num AFND o tempo de reconhecimento é $O(|r| \times |x|)$ e o espaço ocupado é $O(|r|)$.
- Num AFD o tempo de reconhecimento é $O(|x|)$ e o espaço ocupado é $O(2^{|r|})$.
- Contudo, num AFD o número de estados pode crescer exponencialmente, por exemplo, em $(a|b)^*a(a|b)(a|b)(a|b) \dots (a|b)$ se existirem $n-1$ ocorrências de $(a|b)$ são necessários 2^n estados.
- Através de *lazy transition evaluation* os estados e as transições são calculados durante a execução, sendo apenas guardadas as transições usadas. Este método é complexo mas supera os anteriores em espaço e tempo.

Analizador lexical

Um analisador lexical produz os elementos lexicais de um programa com base numa linguagem regular:

modularidade: permite separar a sintaxe em duas fases distintas: análise lexical e sintática.

legibilidade: expressões regulares são, em geral, mais legíveis.

simplicidade: permite simplificar significativamente o analisador sintático.

eficiência: separação lexical e sintática permite que ambos os analisadores sejam mais eficientes (usa autômato sem pilha auxiliar).

portabilidade: variações entre ambientes, dispositivos ou sistemas operativos podem ficar contidos no analisador lexical.

Tarefas do analisador lexical

- Identificação de elementos lexicais (tokens): literais, palavras reservadas, identificadores, operadores, separadores e delimitadores.
- Tratamento de caracteres brancos e comentários.
- Manipulação dos atributos de alguns dos elementos lexicais.
- Utilização da tabela de símbolos para classificar e guardar informação auxiliar de alguns elementos lexicais.
- Análise do grafismo (posicionamento do elementos em linha e coluna) do programa.
- Processamento de macros.
- Tratamento de erros, em geral ignorando os caracteres, embora este tipo de erros seja raro.

Formato do ficheiro lex

Ficheiro, com a extensão **.l**, dividido em três zonas separadas por uma linha contendo apenas **%%**:

- declarações: de macros, de agrupamentos e declarações da linguagem de apoio entre **%{** e **%}**.
- regras: expressão regular separada da ação semântica por um ou mais espaços brancos. A ação semântica é uma instrução da linguagem ou bloco delimitado por chavetas.
- código: realização de funções, algumas das quais declaradas acima.

Gerar um analisador lexical, designado por **lex.yy.c**, com o comando **lex xxx.l** e compilado com o auxílio da biblioteca **-ll**.
(O **flex** usa a biblioteca **-lfl**)

Expressões regulares

x	O carácter 'x'
"x"	O carácter 'x', mesmo que seja um carácter especial
$\backslash x$	O carácter 'x', mesmo que seja um carácter especial
$x\$$	O carácter 'x' no fim da linha
\hat{x}	O carácter 'x' no início da linha
$x?$	Zero ou uma ocorrência de 'x'
$x+$	Uma ou mais ocorrências de 'x'
x^*	Zero ou mais ocorrências de 'x'
xy	O carácter 'x' seguido do carácter 'y'
$x y$	O carácter 'x' ou o carácter 'y'
$[az]$	O carácter 'a' ou o carácter 'z'
$[a-z]$	Do carácter 'a' ao carácter 'z'
$[\^a-z]$	Qualquer carácter exceto de 'a' a 'z'
$x\{n\}$	'n' ocorrências de 'x'
$x\{m,n\}$	de 'm' a 'n' ocorrências de 'x'
x/y	'x' se seguido por 'y' (só 'x' faz parte do padrão)
$.$	Qualquer carácter exceto $\backslash n$
(x)	O mesmo que 'x', parenteses alteram a prioridade dos operadores
$<< EOF >>$	Fim do ficheiro

Tratamento de expressões regulares

Identificação da ação semântica a executar, quando mais de uma expressão regular é válida:

- A sequência de entrada mais comprida é a escolhida.
- Em caso de igualdade de comprimento é usada a que se encontra primeiro no ficheiro de especificação.

Notar que não se trata da expressão regular maior, mas da sequência de entrada maior:

```
%%
dependente      printf("Encontrei 'dependente'\n");
[a-z]+          ECHO;
```

Funções

int yylex(void) : rotina, gerada pelo **lex**, que realiza a análise lexical. Devolve o número do elemento lexical encontrado ou 0 (zero) quando atinge o fim do processamento.

int yywrap(void) : rotina, escrita pelo programador, que quando um ficheiro chega ao fim permite continuar o processamento noutro ficheiro. Caso não haja mais ficheiros a processar **yywrap()** devolve 1 (um), caso contrário atualiza a variável **yyin** para o ficheiro seguinte e devolve 0 (zero).

void yymore(void) : rotina, invocada numa ação semântica, que permite salvar guardar o texto reconhecido pela expressão regular para seja concatenado com a expressão regular seguinte.

void yyless(int n) : rotina, invocada numa ação semântica, que permite considerar apenas os primeiros **n** caracteres de **yytext**, sendo os restantes reconsiderados para processamento.

Variáveis globais

char yytext[] : cadeia de caracteres que contém o texto reconhecido pela expressão regular.

int yyleng : comprimento da cadeia de caracteres que contém o texto reconhecido.

int yylineno : número de linha do ficheiro de entrada onde se encontra o último carácter reconhecido pela expressão regular. Em **flex** usar a opção **-l** ou incluir **%option yylineno** ou **%option lex-compat** no ficheiro **.l**.

FILE *yyin : ponteiro para o ficheiro de onde são lidos os caracteres a analisar.

FILE *yyout : ponteiro para o ficheiro de onde é escrito o texto através da macro **ECHO**, ou outro texto que o programador deseje.

YYSTYPE yylval : variável que transporta o valor do elemento lexical reconhecido para outra ferramenta.

Macros predefinidas

ECHO : imprime o texto reconhecido (ou seja, **yytext**) pela expressão regular, ou acumulado de outras regras através de sucessivas invocações a **yytext()**. Na realidade está definido como

```
#define ECHO fwrite(yytext, yyleng, 1, yyout)
```

REJECT : depois de processada a ação semântica que inclui a chamada ao **REJECT** o processamento recomeça no início do texto reconhecido pela regra mas ignorando a regra atual.

Acesso direto a funções de entrada/saída

int input(void) : esta rotina permite ler o carácter seguinte, a partir do ficheiro de entrada, sem que seja processado pelo analisador lexical. O valor -1 (fim de ficheiro) é apenas devolvido no fim do processamento, pois a rotina **yywrap()** é chamada se necessário.

void output(int) : imprime o carácter em **yyout**. Esta rotina não é suportada pelo flex.

void unput(int) : recoloca o carácter passado como argumento para processamento pelas expressões regulares seguintes. Notar que caso se pretenda recolocar diversos caracteres este devem ser recolocados pela ordem inversa.

Substituições

- As substituições permitem simplificar a escrita as expressões regulares.
- Expressão regular, na zona das declarações seguida do identificador da substituição.
- Usa-se, em expressão regulares subsequentes, entre chavetas.

```
DIG [0-9]
INT {DIG}+
EXP [Ee][+-]?{INT}
REAL {INT}"."{INT}({EXP})?
```

Agrupamentos

- Grupos de expressões regulares ativadas por ações 'BEGIN' e identificadas por '%s' na zona das declarações.
- As expressões regulares do agrupamento são precedidas do identificador entre < e >. O agrupamento 'INITIAL' identifica as regras globais, permanentemente ativas.
- Em cada instante apenas estão ativas as regras globais e um dos agrupamentos, se tiver sido executada uma ação 'BEGIN'.

```
%s IN
%%
<IN>.\n ECHO;
<IN>^"%%" BEGIN INITIAL;
^"%%" BEGIN IN;
.\n ;
```

Ligação ao yacc

elementos a ignorar : comentários ou espaços brancos, por exemplo.

elementos lexicais úteis (tokens) : descritos por

tipo : número inteiro devolvido (instrução `return` na ação semântica) pela rotina `yylex()` e que descreve o *token* encontrado: valores de 1 a 255 para caracteres isolados ASCII e > 256 para conjuntos de caracteres. (devolve 0 para fim e 256 para erro)

valor : quantidade a guardar na variável global `yyval` para alguns *tokens*, por exemplo inteiros, identificadores ou cadeia de caracteres.

descrição dos tokens necessários : produzido pelo `yacc`, com a opção `-d`, no ficheiro `y.tab.h`. Contém as constantes > 256 a variável `yyval` e seu tipo, devendo ser incluído nas declarações.

Extensões do flex

modo mais compatível com *lex* : gerar com `flex -l` ou incluir `“%option lex-compat”` nas declarações.

acesso a *yylineno* : usar o modo de compatibilidade com `lex` ou incluir `“%option yylineno”` nas declarações.

agrupamentos exclusivos : apenas as regras do agrupamento ativo estão válidas, não incluindo as globais, usando `“%x”` em vez de `“%s”`.

modo *debug* : gerar com `flex -d` e colocar a variável `yy_flex_debug` a 1.

Eficiência de processamento

- tempo de processamento do autómato proporcional à dimensão do ficheiro a processar e não ao número de expressões regulares usadas (pode influir no número de estados e logo na espaço ocupado).
- utilizar o mais possível expressões regulares e fazer o mínimo em `C`.
- regras mais específicas no princípio da especificação (palavras reservadas, por exemplo) e regras mais genéricas no fim especificação (identificadores, por exemplo).

Definição e compilação de uma linguagem

- Uma linguagem é um alfabeto e um conjunto de frases.
- Uma linguagem pode ser definida por:
 - sintaxe**: gramática livre de contexto, BNF (Backus-Naur Form).
 - semântica**: informal (textual), operacional, denotacional, por ações.
- Compilação de um programa descrito numa linguagem:
 - objetivo**: traduzir expressões infixadas em postfixadas.
 - aplicabilidade**: computação baseada em pilha de dados.

Gramática como descrição sintática

- Uma gramática dá uma explicação precisa e fácil de compreender da sintaxe de uma linguagem.
- Para algumas classes de gramáticas é possível gerar automaticamente um analisador sintático eficiente.
- Analisadores da gramática identificam ambiguidades e outras construções difíceis de serem reconhecidas e que passam despercebidas.
- A gramática tem relação direta com a estrutura da linguagem usada, facilitando a análise, tradução e compilação.
- Uma gramática é de fácil extensão e atualização, garantido maior durabilidade do analisador sintático.

Notação BNF: Backus-Naur Form

- um conjunto de símbolos **terminais** (tokens), onde cada um representa um elemento lexical extraído do programa a analisar.
- um conjunto de símbolos **não terminais**, onde cada símbolo não terminal é expandido em pelo menos uma produção.
- um conjunto de **produções**, onde cada produção representa a expansão de numa sequência de símbolos numa outra sequência de símbolos, eventualmente vazia:
 - uma produção: $\alpha\beta \rightarrow \gamma\delta$
 - produções alternativas $\alpha \rightarrow \beta_1, \alpha \rightarrow \beta_2, \dots, \alpha \rightarrow \beta_n$ são representadas por $\alpha \rightarrow \beta_1 | \beta_2 | \dots | \beta_n$.
 - uma produção nula: $\alpha \rightarrow \varepsilon$.
- um dos símbolos não terminais é designado por símbolo **inicial**.

Gramática livre de contexto

- Uma gramática livre de contexto possui apenas produções com um só símbolo não terminal do lado esquerdo.
- Tudo o que pode ser descrito por uma expressão regular pode ser descrito por uma gramática livre de contexto.
- Extensões ao BNF (EBNF):
 - $\{ \}$ para representar zero ou mais repetições de símbolos.
 - $[]$ para representar elementos opcionais.
 - $()$ para agrupar símbolos.
 - $a +$ para representar uma ou mais repetições de símbolos.
 - a / b para representar repetições de **a** através de **b**.

Métodos de análise gramatical

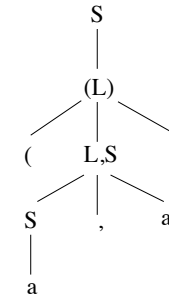
- Métodos de *parsing* universais: funcionam para qualquer gramática, mas são muito ineficientes (inviáveis para uso prático).
- Métodos de processamento determinístico com leitura da esquerda para a direita (L?), um símbolo de cada vez:
 - descendentes (LL ou top-down):** constroem as árvores sintáticas da raiz para as folhas.
 - ascendentes (LR ou bottom-up):** constroem as árvores sintáticas das folhas para a raiz.
- Estes métodos funcionam para um conjunto limitado de gramáticas, que na prática são suficientes para a maioria das linguagens usadas em computação (em particular as linguagens de programação).

Derivações

- Considerando a gramática:
$$\begin{array}{l|l} S & \rightarrow (L) \\ L & \rightarrow L, S \end{array} \quad \begin{array}{l} a \\ S \end{array}$$
- Derivação: substituição de uma sequência de símbolos, que estejam do lado esquerdo de uma produção, pelos símbolos do lado direito de uma produção.
- Derivação mais à esquerda: expande-se o não terminal mais à esquerda. Por exemplo, $S \Rightarrow^1 (L) \Rightarrow^3 (L, S) \Rightarrow^4 (S, S) \Rightarrow^2 (a, S) \Rightarrow^2 (a, a)$
- Derivação mais à direita: expande-se o não terminal mais à direita. Por exemplo, $S \Rightarrow^1 (L) \Rightarrow^3 (L, S) \Rightarrow^2 (L, a) \Rightarrow^4 (S, a) \Rightarrow^2 (a, a)$
- Derivação imediata: $S \Rightarrow (L)$
- Derivação de um ou mais passos: $S \Rightarrow^+ (a, a)$

Árvores sintáticas

- Árvores sintáticas são representações gráficas das derivações:



nó: símbolo não terminal.

ramo: derivação.

folha: símbolo terminal.

- A construção da árvore pode ser descendente ou ascendente.

Recursividade e ambiguidade

- Uma gramática diz-se recursiva se existir:
 - uma derivação recursiva à esquerda: $\alpha \Rightarrow^+ \alpha\beta$.
Por exemplo, $num \rightarrow num\ digit$.
 - uma derivação recursiva à direita: $\alpha \Rightarrow^+ \beta\alpha$.
Por exemplo, $num \rightarrow digit\ num$.
- Uma gramática diz-se ambígua se produz mais de uma árvore sintática, para a mesma frase.
Por exemplo, para a gramática $E \rightarrow E - E \mid int$ existem duas árvores sintáticas que resultam de derivações mais à esquerda da frase $3 - 5 - 7$:
 - $E \Rightarrow E - E \Rightarrow E - E - E \Rightarrow int_3 - E - E \Rightarrow int_3 - int_5 - E \Rightarrow int_3 - int_5 - int_7$, ou seja, $(3 - 5) - 7 = -9$.
 - $E \Rightarrow E - E \Rightarrow int_3 - E \Rightarrow int_3 - E - E \Rightarrow int_3 - int_5 - E \Rightarrow int_3 - int_5 - int_7$, ou seja, $3 - (5 - 7) = 5$.

Prioridade e associatividade

- Cada nível de prioridade inclui os operadores de igual prioridade e os símbolos não terminais associados à mesma prioridade ou superior.
- A associatividade é definida pela recursividade das produções.
- Por exemplo,

$E \rightarrow E + T$	$E - T$	T
$T \rightarrow T * F$	T / F	F
$F \rightarrow U ** F$	U	
$U \rightarrow -P$	P	
$P \rightarrow (E)$	int	id

Conjuntos FIRST

- O conjunto FIRST de um símbolo não terminal é o conjunto de símbolos terminais situados mais à esquerda em todas as derivações possíveis. O símbolo ε só pertence ao conjunto FIRST de α se existir uma derivação $\alpha \Rightarrow^+ \varepsilon$.

- $X \in FIRST(\alpha)$ se $\alpha \rightarrow X\beta \wedge X$ terminal.
- $(FIRST(\beta) \setminus \{\varepsilon\}) \subset FIRST(\alpha)$ se $\alpha \rightarrow \beta X \wedge \beta$ não terminal.
- $\varepsilon \in FIRST(\alpha)$ se $\alpha \Rightarrow^+ \varepsilon$.
- $FIRST(\beta) \subset FIRST(\alpha)$ se $\alpha \rightarrow X\beta \wedge X \Rightarrow^+ \varepsilon$.

- Por exemplo, $FIRST(S) = \{a, b, c, d, \varepsilon\}$ em:

$$\begin{aligned} S &\rightarrow a|Xb|XY \\ X &\rightarrow Xc|\varepsilon \\ Y &\rightarrow dY|\varepsilon \end{aligned}$$

Conjuntos FOLLOW

- O conjunto FOLLOW de um símbolo não terminal é o conjunto de símbolos terminais derivados imediatamente após a identificação desse símbolo não terminal.

- $\$ \in FOLLOW(S)$ se S é o símbolo inicial.
- $(FIRST(\beta) \setminus \{\varepsilon\}) \subset FOLLOW(\alpha)$ se $X \rightarrow \alpha\beta$.
- $FOLLOW(\alpha) \subset FOLLOW(\beta)$ se $\alpha \rightarrow X\beta$.
- $FOLLOW(\alpha) \subset FOLLOW(\beta)$ se $\alpha \rightarrow \beta X \wedge X \Rightarrow^+ \varepsilon$.

- Por exemplo, $FOLLOW(Y) = \{a, b, \$\}$ em:

$$\begin{aligned} S &\rightarrow SaYX|c \\ X &\rightarrow bX|\varepsilon \\ Y &\rightarrow 0|1 \end{aligned}$$

Conjuntos LOOKAHEAD

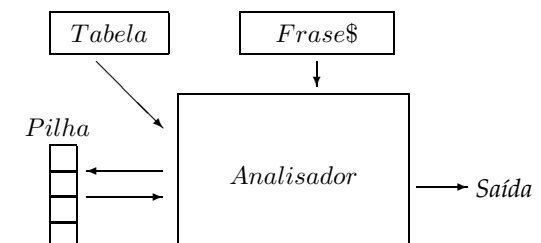
- A relação LOOKAHEAD permite identificar os símbolos terminais que permitem a expansão da regra:

$$LOOKAHEAD(A \rightarrow \alpha) = FIRST(A) \cup (FOLLOW(\alpha) \wedge \alpha \Rightarrow^+ \varepsilon)$$

- A relação LOOKAHEAD permite verificar se a gramática pode ser processada por um analisador preditivo descendente por tabela.

Autômatos de pilha

- linguagens regulares podem ser modeladas por autômatos finitos: a transição para um estado só depende do estado atual e do símbolo de entrada.
- linguagens não regulares necessitam de autômato de pilha: a transição para um estado depende do estado atual, do símbolo de entrada e do conteúdo da pilha.



Gramáticas LL(1)

- Uma gramática independente do contexto diz-se LL(1) se:
 - não possui recursividades à esquerda.
 - em todas as regras com o mesmo símbolo do lado esquerdo, $X \rightarrow \alpha$ e $X \rightarrow \beta$, verifica-se a condição:
 $LOOKAHEAD(X \rightarrow \alpha) \cap LOOKAHEAD(X \rightarrow \beta) = \emptyset$
- Dada uma gramática pode ser possível identificar outra gramática equivalente que seja LL(1), através de transformações como:
 - eliminação da recursividade à esquerda.
 - fatorização à esquerda.

Uma gramática LL(1) pode ser processada por um analisador preditivo descendente por tabela.

Eliminação de produções não atingíveis

- Um símbolo não terminal que nunca consegue ser atingido a partir do símbolo inicial da gramática, é inatingível e, deve ser retirado.
- Se o símbolo não terminal nunca aparece do lado direito das regras então é inatingível, mas pode também haver ciclos de regras inatingíveis.

$$\begin{aligned} S &\rightarrow A \\ A &\rightarrow \alpha \\ B &\rightarrow \beta \\ C &\rightarrow \gamma D \\ D &\rightarrow \delta C \end{aligned}$$

onde apenas as primeiras duas regras são atingíveis.

Fatorização à esquerda

- Para uma gramática poder ser processada por um analisador preditivo descendente por tabela é necessário ser possível identificar qual a regra a expandir quando um símbolo terminal é lido.
 Por exemplo, em $X \rightarrow \alpha\beta_1|\alpha\beta_2$, a leitura de α não permite escolher uma das regras.
- Fatorização à esquerda substitui $X \rightarrow \alpha\beta_1|\alpha\beta_2$ por:

$$\begin{aligned} X &\rightarrow \alpha X' \\ X' &\rightarrow \beta_1|\beta_2 \end{aligned}$$

Substituição de cantos à esquerda

- Numa produção $A \rightarrow B\alpha$ o símbolo A é designado por um canto à esquerda. As produções vazias (ϵ) não têm cantos.
- Numa substituição de cantos à esquerda, os símbolos não terminais que sejam cantos à esquerda de uma produção podem ser substituídos pelas produções do símbolo não terminal em questão.

Por exemplo,

$$\begin{aligned} A &\rightarrow \beta|B\alpha \\ B &\rightarrow \beta|\gamma \end{aligned}$$

produz

$$A \rightarrow \beta|\beta\alpha|\gamma\alpha$$

necessitando ainda de fatorização.

Substituição de singularidades

- Se um símbolo não terminal deriva uma só regra, então pode ser substituído nas regras onde é referido.
- Estas produções singulares (uma só regra) reduzem a dimensão da gramática e respectivas tabelas, bem como menor número de passos na análise das sequências de entrada.

$$A \rightarrow \alpha B \beta B \gamma$$

$$B \rightarrow \delta$$

resume-se a

$$A \rightarrow \alpha \delta \beta \delta \gamma$$

Eliminação de ambiguidade

- Produções ambíguas são aquelas que possuem mais de uma ocorrência de um determinado símbolo não terminal do lado direito da produção.
- A ambiguidade pode ser retirada dando precedência a uma das ocorrências face às outras.

Numa regra $A \rightarrow \alpha A \beta A \gamma | \alpha_1 | \dots | \alpha_n$ optando pela associatividade

$$\begin{array}{l} \text{à direita} \quad A \rightarrow \alpha A' \beta A \gamma | A' \\ \quad \quad \quad A' \rightarrow \alpha_1 | \dots | \alpha_n \end{array} \quad \text{ou à esquerda} \quad \begin{array}{l} A \rightarrow \alpha A \beta A' \gamma | A' \\ A' \rightarrow \alpha_1 | \dots | \alpha_n \end{array}$$

- Se a gramática inclui várias regras ambíguas, eliminar a ambiguidade passo a passo começando pelas regras de menor prioridade.

Eliminação da recursividade à esquerda

- Para uma gramática poder ser processada por um analisador preditivo descendente por tabela não podem existir derivações à esquerda $A \Rightarrow^+ A\alpha$.
- Para eliminar recursividades à esquerda substitui-se as regras do tipo $A \rightarrow A\alpha | \beta$ por:

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' | \varepsilon$$

Construção da tabela de análise

- A tabela de análise de um analisador preditivo descendente por tabela é construída por uma coluna por cada símbolo terminal mais o símbolo \$, e por uma linha por cada símbolo não terminal.
- Cada regra $A \rightarrow \alpha$ da gramática é incluída na tabela em:
 - $tab[A, a] : \text{se } a \in FIRST(\alpha)$.
 - $tab[A, b] : \text{se } \varepsilon \in FIRST(\alpha) \wedge b \in FOLLOW(A)$.
 - $tab[A, \$] : \text{se } \varepsilon \in FIRST(\alpha) \wedge \$ \in FOLLOW(A)$.

Todas os elementos não preenchidos da tabela correspondem a um erro.

Exemplo de tabela LL(1)

- Eliminação da recursividade:

$$\begin{array}{lcl} S \rightarrow (L) \mid a & & S \rightarrow (L) \mid a \\ L \rightarrow L, S \mid S & \longrightarrow & L \rightarrow SL' \\ & & L' \rightarrow , SL' \mid \varepsilon \end{array}$$

- Determinação dos conjuntos FIRST, FOLLOW e LOOKAHEAD:

	FIRST	FOLLOW	LOOKAHEAD
$S \rightarrow (L) \mid a$	(a	\$,)	(a
$L \rightarrow SL'$	(a)	(a
$L' \rightarrow , SL' \mid \varepsilon$, ε)	,)

- Construção da tabela:

	()	,	a	\$
S	$S \rightarrow (L)$			$S \rightarrow a$	
L	$L \rightarrow SL'$			$L \rightarrow SL'$	
L'		$L' \rightarrow \varepsilon$	$L' \rightarrow , SL'$		

Análise preditiva por tabela

- Considerando X o símbolo no topo da pilha e a o símbolo de entrada:
 - se $X = a = \$$ então o analisador chegou ao fim de uma sequência correta.
 - se $X = a \neq \$$ então o analisador retira o símbolo X do topo da pilha e avança para o símbolo seguinte na sequência de entrada.
 - se X é terminal e $X \neq a$ então trata-se de um erro sintático.
 - se X é não terminal então a regra em $tab[X, a]$ é utilizada:
 - * se $tab[X, a]$ estiver vazia então trata-se de um erro sintático.
 - * se $tab[X, a]$ contiver $X \rightarrow UVW$ então os símbolos do lado direito da regra são colocados na pilha, com U no topo.

Exemplo das ações do analisador LL(1)

pilha	entrada	regra
$S\$$	$(a, a)\$$	$L \rightarrow (L)$
$(L)\$$	$(a, a)\$$	$(\equiv ($
$L)\$$	$a, a)\$$	$L \rightarrow SL'$
$SL')\$$	$a, a)\$$	$S \rightarrow a$
$aL')\$$	$a, a)\$$	$a \equiv a$
$L')\$$	$, a)\$$	$L' \rightarrow , SL'$
$, SL')\$$	$, a)\$$	$, \equiv ,$
$SL')\$$	$a)\$$	$S \rightarrow a$
$aL')\$$	$a)\$$	$a \equiv a$
$L')\$$	$)\$$	$L' \rightarrow \varepsilon$
$)\$$	$)\$$	$) \equiv)$
$\$$	$\$$	$\$ \equiv \$$ (accept)

Recuperação de erros em analisadores LL(1)

conjunto de sincronização onde os membros do conjunto são símbolos que terminam blocos de código:

1. retirar elementos da pilha até ficar no topo um dos elementos do conjunto de sincronização. A recuperação não é possível se a pilha não contiver elementos do conjunto;
2. ler elementos lexicais até o símbolo de antevisão coincidir com o topo da pilha. Se estiver no fim do ficheiro, a recuperação falhou, caso contrário o processamento continua.

pilha de sincronização auxiliar onde conjuntos de sincronização, definidos regra a regra, são colocados quando a regra é derivada. Simultaneamente, retira-se o símbolo a derivar, coloca-se uma marca e os símbolos derivados na pilha do analisador. Quando essa marca fica no topo da pilha, retira-se a marca e o conjunto de sincronização. Em caso de erro, usa-se o conjunto de sincronização no topo da pilha auxiliar.

Recuperação de erro nas ações do analisador LL(1)

Sync = { } }

pilha	entrada	regra
S\$	(, a)\$	$L \rightarrow (L)$
(L)\$	(, a)\$	$(\equiv ($
L)\$, a)\$	(syntax error)
)\$)\$	$) \equiv)$
\$	\$	$\$ \equiv \$$

Compressão de tabelas: double-offset indexing

- Arrumam-se as linhas da tabela(t) original num vetor(v), de tal forma que não haja sobreposição de elementos não nulos, minimizando o número de posições nulas. Heurística (problema NP-completo): começar pelas linhas de maior densidade.
- Cada posição no vetor contém o valor e a linha original a que pertence.
- Uma tabela de coluna(c) indica o início de cada coluna no vetor.
- Se $v[c[i] + j].line == i$ então $v[c[i] + j].value$ contém o valor da tabela original, caso contrário é um elemento nulo.

L			P	
	Q			R
		U		
W	X			

 $= c$

2	2	2	0
---	---	---	---

 $+ v$

W	X	L	Q	U	P	R
4	4	1	2	3	1	2

Analísadores LL(1) *hardcoded*

- A codificação explícita de analisadores descendentes preditivos é simples e usa a pilha do processador para guardar o caminho.

```
static void symbL2() {
    if (la == ',') { la = yylex(); symsS(); symbL2(); return; }
    if (la == ')') { return; }
    error();
}
```

- A recuperação de erro é possível, com recurso a exceções (`long jmp`/`set jmp`), mas mais complicada de realizar.
- A avaliação de atributos sintetizados (ver a seguir) é simples e direta, mas a utilização de atributos herdados exige uma pilha auxiliar.

Analísadores LL(k) com $k > 1$

- A utilização de mais de um símbolo de antevisão permite superar alguns dos problemas das gramáticas LL(1).
- Os conjuntos $FIRST_k$, $FOLLOW_k$ e $LOOKAHEAD_k$ são calculados utilizando k símbolos terminais (*tokens*) para todos símbolos não-terminais (NT) da gramática.
- Por exemplo, $FIRST_2$ será constituído pelas possíveis combinações de sequências de 2 símbolos de entrada, para a regra em análise. Ou seja, se $w \rightarrow 'a'x|'a'y|z$ pode ser $FIRST_2(regra) = \{ 'a' 'b', 'a' 'c', 'z' 'b' \}$.
- Contudo, a dimensão tabela de análise tem um crescimento exponencial pois é $NT \times tokens^k$, pelo que se torna impraticável para gramáticas vulgares.

Analísadores Strong LL(k)

- Como nem todas as regras de uma gramática necessitam de mais de um símbolo de antevisão, a solução passa por utilizar mais símbolos de antevisão apenas quando necessário.
- A determinação da regra lê o número necessário de símbolos de antevisão até determinar a produção a selecionar. O processamento da produção selecionada é que vai, efetivamente, consumir os símbolos anteriormente lidos, tal como no caso LL(1).
- A determinação da regra na solução *hardcoded* é efetuada através de sequências de instruções *switch* aninhadas, em os *case* incluem apenas as soluções possíveis (e o erro é detetado pelo *default*. (ferramenta *antlr*)
- A determinação da regra pode recorrer a um AFD, onde as transições correspondem a possíveis sequências de símbolos e os estados terminais identificam as regras a selecionar. Uma marca na tabela de análise LL(1) identifica, em certas combinações $[NT, token]$ que necessitam de $k > 1$, não um *token* mas um AFD.

Gramáticas atributivas

- **Gramática atributiva:** gramática independente do contexto onde símbolos (terminais e não terminais) recebem valores manipulados por regras semânticas.
- **Regras semânticas:** são funções da forma $b = f(c_1, c_2, \dots, c_n)$, onde os c_i são literais, valores de atributos dos símbolos da produção ou funções puras (sem efeitos colaterais).
- **Atributo sintetizado:** na regra $A \rightarrow \alpha$ o valor b é um atributo sintetizado de A .
- **Atributo herdado:** na regra $A \rightarrow \alpha$ o valor b é um atributo herdado de α , se os c_i dependem de atributos de A ou de α .

Exemplo de gramática atributiva

$$\begin{array}{lll}
 num & \rightarrow & seq \quad \{ \quad seq.pos = 0; \\
 & & \quad \quad num.val = seq.val; \} \\
 seq & \rightarrow & seq_1 \, dig \quad \{ \quad seq_1.pos = seq.pos + 1; \\
 & & \quad \quad dig.pos = seq.pos; \\
 & & \quad \quad seq.val = seq_1.val + dig.val; \} \\
 & & | \quad dig \quad \{ \quad dig.pos = seq.pos; \\
 & & \quad \quad seq.val = dig.val; \} \\
 dig & \rightarrow & 0 \quad \{ \quad dig.val = 0; \} \\
 & & | \quad 1 \quad \{ \quad dig.val = 2^{dig.pos}; \}
 \end{array}$$

- não-terminal *dig* tem 2 atributos:
 - *pos* representa a posição do dígito no número binário (herdado *dig.pos*).
 - *val* representa o valor do dígito no número binário (sintetizado *dig.val*).
- atributos com o mesmo significado em *seq* (*pos* e *val*) e em *num* (*val*).

Avaliação dirigida pela sintaxe

- **A avaliação é dirigida pela sintaxe** quando os valores dos atributos são calculados quando se identifica a derivação das regras sintáticas.
 - + análise mais simples e rápida, num único passo.
 - + poupa memória, pois não exige a construção de uma árvore sintática.
 - só se pode aplicar em gramáticas simples.
 - pode obrigar a modificações significativas na gramática.
- **Definição de S-atributos:** gramática atributiva onde todos os atributos são sintetizados.
- **Definição de L-atributos:** gramática atributiva onde os atributos são sintetizados ou herdados “*dos irmãos mais velhos*” (na regra $A \rightarrow \alpha_1 \alpha_2 \alpha_3 \alpha_4$ os atributos de α_3 podem depender dos atributos de α_1, α_2 ou A , mas não de α_4).

Grafos de dependências

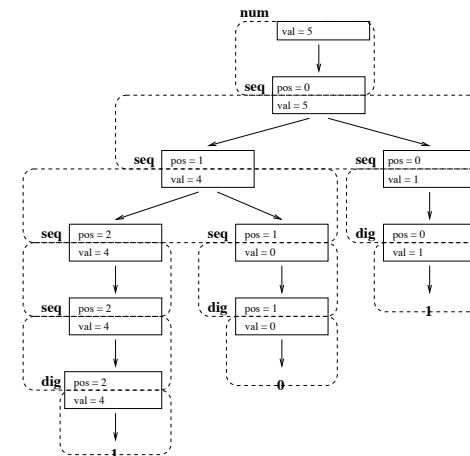
Árvore sintática anotada: árvore sintática onde os nós incluem os atributos e seus valores.

Grafo de dependências: grafo dirigido acíclico que determina a ordem de avaliação dos atributos.

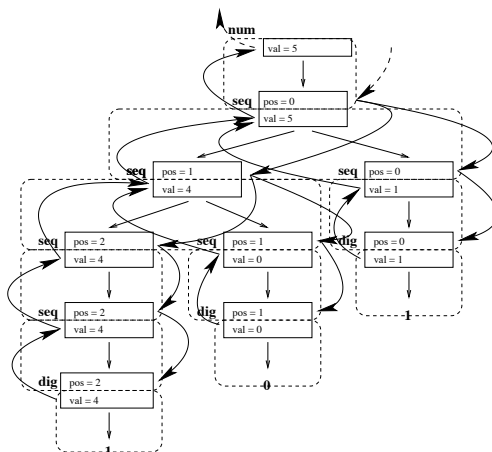
Avaliação de atributos:

- Avaliam-se, primeiro, os atributos herdados, descendo a árvore.
- Avaliam-se, seguidamente, os atributos sintetizados, subindo a árvore.

Árvore sintática anotada



Grafos de dependências



Eliminação de atributos herdados

Identities algébricas: utilizações de expressões equivalentes que, usando apenas atributos sintetizados, permitem obter o mesmo resultado final.

Por exemplo, nos números binários: $2^n = 2 * 2^{n-1}$.

Alteração sintática: os símbolos que definem o valor do atributo herdado são deslocados para produções que o consomem.

Por exemplo, declaração de variáveis em C.

Deslocamento de regras semânticas: atrasar a avaliação de atributos herdados através de estruturas semânticas auxiliares, em geral, árvores sintáticas. (Deixa de poder ser considerado como “dirigido pela sintaxe”)

Por exemplo, declaração de variáveis em Pascal.

(Generalizável através da utilização de árvores sintáticas)

Esquemas de avaliação de atributos

descendente: eliminação da recursividade à esquerda obriga a introduzir:

- um atributo herdado transmite os atributos dos símbolos que faziam parte da mesma regra.
- um atributo sintetizado transmite o resultado na direcção da raiz.

ascendente: definições de L-atributos em analisadores ascendentes:

acções internas: são equivalentes a regras vazias, pelo que reduzem a antevisão do analisador. ($i \rightarrow IF \{ \dots \} ' ' \mid IF y ' ')$)

atributos herdados na pilha: quando um analisador LR explora Y , na regra $A \rightarrow X Y$, os atributos de X já estão na pilha, podendo ser utilizados na regra $Y \rightarrow \dots$. Contudo, Y pode aparecer em outras regras não precedido por X .

Atributos na eliminação da recursividade à esquerda

- Considere-se a gramática seguinte onde cada símbolo tem um atributo sintetizado e f e g são funções:

$$\begin{array}{lcl} A & \rightarrow & A_1 Y \quad \{ A.a = g(A_1.a, Y.y); \} \\ & | & X \quad \{ A.a = f(X.x); \} \end{array}$$

- Removendo a recursividade à esquerda ficamos com um novo símbolo R inclui um atributo herdado h e um sintetizado s :

$$\begin{array}{lcl} A & \rightarrow & X \quad \{ R.h = f(X.x); \} \\ & & R \quad \{ A.a = R.s; \} \\ R & \rightarrow & Y \quad \{ R_1.h = g(R.h, Y.y); \} \\ & & R_1 \quad \{ R.s = R_1.s; \} \\ & | & \varepsilon \quad \{ R.s = R.h; \} \end{array}$$

Formato do ficheiro

Formato idêntico ao do ficheiro **lex**, três zonas separadas por `%%`:

- declarações: token, union, type, left, right, nonassoc, start e declarações da linguagem entre `%{'` e `%}'`.
- regras: a produção é separada por `:'` (em vez de \rightarrow), as alternativas por `'|'` e as ações semânticas são incluídas entre chavetas.
- código: realização de funções, algumas das quais declaradas acima.

Declarações:

- **%start:** declara o símbolo não terminal objetivo (o símbolo inicial) da gramática. Caso esta declaração seja omitida, é assumido o primeiro símbolo na zona das regras.

Elementos lexicais, valores e tipos

- **%token:** declara os elementos lexicais, não constituídos por caracteres isolados, a ser utilizados na gramática como símbolos terminais. Podem existir diversas declarações **%token**, sem ordem relevante, para declarar todos os elementos lexicais.
- **%union:** define os tipos de dados a ser guardados na pilha do analisador sintático. Os dados são associados. quer a símbolos terminais (transportados do analisador lexical), quer a símbolos não terminais.
- **%type< x >:** associa um tipo de dados aos símbolos declarados. O `'x'` representa o nome da variável na union.
- **%token< x >:** declara o token e respetivo tipo numa só declaração.

Prioridade e associatividade

- Embora possa ser resolvida gramaticalmente, a prioridade e associatividade das regras podem ser descritas por declarações:
- **%left**: lista de símbolos que designam regras (aparecem nessas regras) com operadores associativos à esquerda.
- **%right**: idem, mas associativo à direita.
- **%nonassoc**: idem, mas não associativo.
- A ordem das declarações define as prioridades relativas, das menos prioritárias (acima), para as mais prioritárias (abaixo).
- Todos os símbolos na mesma diretiva designam regras com a mesma prioridade.

Ações semânticas

Os valores dos atributos são referidos pela posição dos respetivos símbolos na regra:

- **\$\$**: designa o símbolo objetivo da regra, símbolo do lado esquerdo.
- **\$1**: designa o primeiro símbolo do lado direito da regra. Os restantes símbolos são numerados por ordem crescente.
- **\$< type > 0**: designa o primeiro símbolo associado a um atributo herdado. O tipo é necessário pois o símbolo não existe na regra e pode não ser univocamente dedutível. Os restantes símbolos na pilha são designados por valores negativos.
- Ações internas podem devolver valores que podem ser utilizados em ações semânticas posteriores, na mesma regra, e ocupam uma posição na regra. Por exemplo, $X \rightarrow Y\{\$ = 3; \} Z\{\$ = \$1 * \$2 + \$3\}$.

Ambiente do analisador sintático

- O analisador sintático é realizado pela rotina **int yyparse()**, que devolve 0 se não foram encontrados erros sintáticos e devolve 1, caso contrário.
- Sempre que é encontrado um erro sintático, a função **int yyerror(char*s)**; é chamada pelo analisador e a variável **int yynerrs**; incrementada.
- As variáveis **int yystate**; e **int yychar**; designam o estado atual do analisador de acordo com a tabela LALR(1) e o código do elemento lexical corrente.
- A ferramenta **yacc** gera o ficheiro **y.tab.c** contendo o código do analisador sintático e respetivas tabelas. As opções **-d** e **-v** permitem gerar os ficheiros **y.tab.h** (para ligação ao **lex**) e **y.output** (tabela LALR(1) legível).
- Na geração do compilador, produz-se primeiro o analisador sintático (com o **yacc**) e depois o analisador lexical (com o **lex**). Na edição de ligações (*link*), indica-se primeiro a biblioteca do **yacc**, designada por **-ly**, e depois a biblioteca do **lex**, designada por **-ll** (ou **-lfl** no caso do **flex**).

Tratamento de erros de sintaxe

- Relatar a presença de erros sintáticos com clareza e precisão.
- Permitir a recuperação de situações de erro, por forma a permitir a identificação de mais erros.
- O processamento de erros não deve prejudicar significativamente o processamento de programa corretos, pelo compilador.
- A introdução de regras gramaticais específicas para tratamento de erros é o procedimento mais comum.

Tratamento de erros em YACC

- O símbolo não-terminal (reservado) **error** permite absorver situações de erro, consumindo os símbolos terminais até encontrar um dos seus **follows**.
- A macro **YYERROR** pode ser utilizada, nas ações semânticas, para gerar erros sintáticos semanticamente.
- A função **yyerror(char*s)**; é chamada sempre que é tratado um erro sintático, mesmo quando originário da semântica por **YYERROR**.
- A macro **yyerrok** pode ser utilizada, nas ações semânticas, para continuar o processamento após um erro sintático.
- A macro **yyclearin** pode ser utilizada, nas ações semânticas, para forçar o analisador a reler o elemento lexical de *lookahead*, caso na recuperação de erros se tenha manipulado o ficheiro de entrada (por exemplo, em utilizações interativas).

Depuração gramatical (*debug*)

- **Análise estática:** gerar o analisador sintático com a opção **-v** (`yacc -v gram.y`) e analisar o ficheiro **y.output** que contém os estados do analisador e as transições com os respetivos símbolos.
- **Análise dinâmica:** compilar o analisador sintático com a constante **YYDEBUG** (`gcc -c -DYYDEBUG y.tab.c`, ou `#define YYDEBUG 1` na gramática) e colocar a variável **yydebug** a **1** (um) no início da execução do programa (no main). Notar que só se garante que a variável exista se **YYDEBUG** estiver definido.

Conflitos na gramática LALR(1)

- determinar os itens das regras determinam os estados da gramática. Em $X \rightarrow AB$ existem 3 itens: $X : .AB$, $X : A.B$ e $X : AB.$, onde o ponteiro **'** representa a posição corrente no processamento da regra.
- pode existir mais de um ponteiro em cada instante, mas se no momento da redução de um ponteiro (último item da regra) existirem mais ponteiros com o mesmo símbolo de antevisão, existe um conflito.
- os conflitos são reportados aos pares e podem ser:
 - redução/redução:** os dois ponteiros estão no fim das respetivas regras.
 - deslocamento/redução:** um dos ponteiros está no fim da regra, mas o outro ainda está a meio da regra.
- um símbolo de *lookahead* distinto elimina conflitos, em gramáticas LALR(1).

Conflitos de redução/redução

- o YACC reduz a primeira regra no ficheiro e ignora as restantes regras.
 - identificar, no **y.output**, as regras envolvidas no conflito e determinar os símbolos que geram o conflito.
- $$\begin{array}{ll}
 S : A|B; & S : A|B|X; \\
 A : X|Y; & \longrightarrow A : Y; \\
 B : X|Z; & B : Z;
 \end{array}$$
- fatorizar as regras comuns:
 - agrupar as regras e resolver a ambiguidade nas ações semânticas (Por exemplo, operações polimórficas)

Conflitos de deslocamento/redução

- o YACC desloca em vez de reduzir.
- identificar, no `y.output`, as regras envolvidas no conflito e determinar os símbolos que geram o conflito.
- introduzir prioridades nas regras. (expressões ou if-else, por exemplo)
- enumerar as diversas alternativas. (chamada a função ou for, por exemplo)
- juntar regras, aplicando a propriedade distributiva, para evitar reduções desnecessárias. (blocos com declarações* e instruções*)

Ações de um analisador ascendente

deslocamento (*shift*): consumo de um elemento lexical (símbolo terminal), colocando na pilha o símbolo e o respetivo estado.

redução (*reduce*): identificação de uma regra, substituindo na pilha todos os símbolos dessa regra e respetivos estados pelo símbolo não terminal reduzido.

salto (*goto*): determinação do estado seguinte após uma redução, colocando-o na pilha.

aceitação (*accept*): sucesso na análise sintática.

erro (*error*): erro na análise sintática.

Exemplo de tabela ascendente

$$\begin{array}{lcl} S & \rightarrow & (L) \mid a \\ L & \rightarrow & L, S \mid S \end{array}$$

	()	,	a	\$	L	S
0	s2			s3			1
1					acc		
2	s2			s3		4	5
3		r2	r2		r2		
4		s6	s7				
5		r4	r4				
6		r1	r1		r1		
7	s2			s3			8
8		r3	r3				

Reconhecimento de uma frase

pilha	entrada	ação
0	(a, a)\$	shift-2
0(2	a, a)\$	shift-3
0(2a3	, a)\$	reduce-2
0(2S	, a)\$	goto-5
0(2S5	, a)\$	reduce-4
0(2L	, a)\$	goto-4
0(2L4	, a)\$	shift-7
0(2L4, 7	a)\$	shift-3
0(2L4, 7a3)\$	reduce-2
0(2L4, 7S)\$	goto-8
0(2L4, 7S8)\$	reduce-3
0(2L)\$	goto-4
0(2L4)\$	shift-6
0(2L4)6	\$	reduce-1
0S	\$	goto-1
0S1	\$	accept

Autômato não determinista LR(0), SLR(1) ou LALR(1)

- Construir a gramática aumentada equivalente, acrescentando a regra $S' \rightarrow S\$, onde S é o símbolo inicial.$
- Os estados do autômato não determinista são os estados da gramática e correspondem aos itens das regras. Em $X \rightarrow AB$ existem 3 itens: $X : .AB$, $X : A.B$ e $X : AB.$, onde o ponteiro $.$ representa a posição no processamento da regra.
- As transições entre estados de uma regra são etiquetadas com o símbolo, terminal ou não terminal, que é analisado entre dois estados consecutivos. Por exemplo, $X : .AB \xrightarrow{A} X : A.B$.
- As transições vazias (ϵ) são construídas de um estado onde o ponteiro precede um não terminal para o primeiro estado de todas as regras que derivam esse não terminal. Por exemplo, $X : .AB \xrightarrow{\epsilon} A : .YZ$.

Transporte dos LOOKAHEAD no fecho- ϵ do LALR(1)

- Sempre que se atinge um item do tipo $X : .Ab$, o LOOKAHEAD b é introduzido nos primeiros itens das regras que derivam A . Notar que se $A : .Bv$ então este item tem como LOOKAHEAD b mas introduz apenas v nos primeiros itens das regras que derivam B .
- Num item do tipo $X : a.B$, os LOOKAHEAD são transportados deste item para os primeiros itens das regras que derivam B .
- Numa transição que consome um símbolo, os LOOKAHEAD são transportados da transição que a antecede para a seguinte, sem alteração.
- Sempre que um estado se repete, i.e. contém os mesmos itens, é necessário verificar se os LOOKAHEAD de todos os seus itens são iguais ao original. Caso não sejam, é necessário adicioná-los e propagá-los a todos os estados dele derivados, que tenham sido determinados entretanto.

Tabela LR(0), SLR(1) ou LALR(1)

- Determinar o autômato determinista a partir do autômato não determinista (com LOOKAHEAD no caso do LALR(1)).
- A tabela tem tantos estados quantos os estados do autômato determinista e tantas colunas quantos os símbolos terminais e não terminais da gramática, incluindo o $\$$.
- Deslocamentos: se $goto(I_0, a) = I_2$, onde I_0 e I_2 são estados do autômato determinista, preencher na linha 0 coluna a da tabela $shift - 2$, ou seja $tab[0, a] = s2$.
- Saltos: se $goto(I_4, S) = I_1$ preencher na linha 4 coluna S da tabela $goto - 1$, ou seja, $tab[4, S] = 1$.
- Reduções: a colocação das reduções depende do método a utilizar: LR(0), SLR(1) ou LALR(1).

Tabela de reduções LR(0), SLR(1) e LALR(1)

- Determinar, para cada regra da gramática, os estados do autômato determinista que correspondem aos estados do autômato não determinista que representam o último item dessa regra.
- Incluir na tabela uma redução dessa regra, nas linhas correspondentes aos estados do autômato determinista calculado acima e nas colunas:
 - LR(0):** preencher a redução em todos os símbolos não terminais, exceto o $accept$ que é preenchido apenas no $\$$.
 - SLR(1):** preencher a redução em todos os $follows$ do símbolo não terminal que deriva essa regra.
 - LALR(1):** preencher a redução apenas nos LOOKAHEAD associados ao último item da regra.
- **Nota:** nestes métodos, um erro pode ser apenas detetado no próximo $shift$.

Utilização determinista de gramáticas ambíguas

Conflitos shift-reduce: a definição de prioridade e associatividade entre **regras**:

- optar pelo *shift* quando o símbolo de antevisão está associado a uma regra de maior prioridade, ou pelo *reduce* no caso contrário.
- nos casos de igual prioridade optar por:
 - shift: se a regra for associativa à direita;
 - reduce: se a regra for associativa à esquerda;
 - error: se a regra não for associativa.

Conflitos reduce-reduce: colocar a regra mais específica primeiro, para que seja preferida em caso de conflito. A regra mais geral será reduzida nos restantes casos.

Se as duas regras forem de igual aplicabilidade, uma delas ficará sempre por reduzir e nunca será utilizada.

Compactação de tabelas LR

- Separar a tabela de deslocamento/redução da tabela de saltos e agrupar as linhas idênticas, usando uma tabela de indireção.
- Reduções unitárias: substitui sequências de reduções unitárias (um só símbolo na produção) por uma só regra, caso não existam ações semânticas nas regras intermédias, dependendo do símbolo de antevisão (utilizado em geral na precedência de operadores).
- Propagação de reduções para uniformizar a tabela (ver a seguir).

Compactação por propagação de reduções

- Os estados que contêm apenas reduções de uma só regra (e erros) podem ser eliminados: os *shift* para esses estados são substituídos por reduções tipo L dessa regra.
- Reduções quase únicas: estados com vários shift e reduções de uma só regra (e erros) passam a ter um comportamento por omissão. As reduções e os erros passam a *default* (valor zero) e o vetor *default* passa a incluir a redução. Para os estados com mais de uma redução, estas não são substituídas, ficando *error* no vetor *default*. Vantagem: a tabela fica mais esparsa, logo mais compactável.
- A substituição de situações de erro por reduções, apenas atrasa a deteção dos erros até ao próximo deslocamento, não comprometendo a correção do analisador.

Analísadores LALR(1) *hardcoded*

- Os analisadores LALR(1) *hardcoded* são 2 a 6 vezes mais rápidos que os baseados em tabelas, mas o código duplica de dimensão. A construção é efetuada a partir das mesmas tabelas depois de compactadas.
- Reduções são codificadas como:

```
reduce_M:
    ação semântica associada à regra M
    stack -= n;
    stack->semantic = yyredval;
    goto nonterminal_P;
```
- As transições dos não-terminais são codificadas como:

```
nonterminal_J:
    switch ((stack-1)->state) {
        case K: goto state_L;
        ...
```

Ações em analisadores LALR(1) *hardcoded*

- As ações baseiam-se no conceito de *shift state* - estados de destino de operações de deslocamento, efetuando o deslocamento no destino. (por exemplo, se existe *s2* então o estado 2 é um *shift state*)

- As ações são codificadas como:

```
state_N:
    stack->state = N;
    stack->semantic = yylval; token = yylex(); // se shift state
    if (++stack == EOS) goto stack_overflow;
state_action_N:
    switch (token) {
        case Q: goto state_X;
        case R: goto reduce_Y;
        case S: goto error_handler;
        case T: YYACCEPT;
        default: goto error_handler; // ou reduce_Z
```

Recuperação de erros: símbolo *erro*

- o utilizador inclui na gramática regras para recuperação de erros em locais específicos, utilizando um símbolo de erro especial nessas regras.
- em caso de erro (acesso a uma posição da tabela sem uma operação válida):
 1. verificar se existe um deslocamento do símbolo de erro para esse estado, caso não exista retirar sucessivamente símbolos da pilha até atingir um estado que desloque o símbolo de erro.
 2. avançar sucessivamente na sequência de entrada até atingir um símbolo de antevisão para exista uma ação que não seja de erro.
 3. se a ação for uma redução é possível que o símbolo de antevisão venha a causar novo erro, pelo que devem ser evitadas regras cuja derivação seja apenas o símbolo de erro.
- cuidado: ações semânticas com efeitos secundários podem ficar por executar.

Recuperação de erros: *panic-mode*

- recuperação de erros em analisadores ascendentes é difícil, pois a pilha contém os elementos lidos, mas não inclui informação sobre o que é esperado. Uma técnica, designada por *panic-mode*, usa a própria tabela:
 1. guardar uma cópia da pilha do analisador;
 2. retirar o elemento do topo e verificar se existe uma entrada válida na tabela. Se existir a recuperação está concluída;
 3. se não existir entrada válida voltar a 2. até esvaziar a pilha;
 4. quando a pilha fica vazia, avançar um elemento lexical, repor uma cópia da pilha inicial (antes do erro) e voltar a 2.;
 5. se se atingiu o fim do ficheiro (último elemento lexical), não foi possível recuperar do erro.
- Devido ao risco de erros em cascata, não imprimir mensagens nos 4 a 5 passos (deslocamentos ou reduções) que se seguem ao erro.

LR canónico: LR(1)

- O autómato LR(1) é construído a partir da gramática aumentada, replicando os itens de cada regra para todos os símbolos de *follow* do símbolo não terminal que deriva.
- As transições vazias (ϵ) são construídas apenas para os itens cujo *follow* corresponde ao *lookahead* desse não terminal no item de origem da transição. Por exemplo, $X : .Ab, \$ \rightarrow^\epsilon A : .YZ, b$, onde $\$$ e b são os *follows* dos itens.
- A tabela de reduções é construída individualizando os símbolos de *follow* de cada regra.
- **Agrupamento de estados LALR(1):** dois estados do analisador LR(1) são agrupados se tiverem os mesmos itens e estes itens diferirem apenas nos símbolos de antevisão.

Analísadores quase LALR(1)

Se gramática for LR(1) mas não for LALR(1) podemos contruir um analisador quase LALR(1):

- Construir o analisador LR(1) e agrupar os estados:
 - ao agrupar dois estados compatíveis podemos forçar outros estados não compatíveis a serem agrupados, provocando um conflito *reduce-reduce*.
 - em caso de gerar conflito fazer *backtrack* e tentar outros agrupamentos.
 - a ordem de agrupamento pode influir no número final de estados (o número mínimo só pode ser obtido depois de tentadas todas as ordenações).
- Agrupar os estados à medida que vão sendo identificados: algoritmo de Pager.

Algoritmo de Pager

Compatibilidade fraca: agrupar 2 estados, s e \bar{s} , com o mesmo núcleo se para todos os pares de conjuntos de antevisão dos seus itens:

- $L_1 \cap \bar{L}_2 = \emptyset$
- $\bar{L}_1 \cap L_2 = \emptyset$
- $L_1 \cap L_2 \neq \emptyset$
- $\bar{L}_1 \cap \bar{L}_2 \neq \emptyset$

Compatibilidade forte: agrupar estados rejeitados pela compatibilidade fraca mas onde o potencial conflito nunca é atingido:

- verificar se as regras, cujos itens incluem os símbolos que geram o conflito, fazem parte dos mesmos estados até à respetiva redução.
- usar o algoritmo de *backtracking* nos estados do analisador quase LALR(1) gerado.

Analísadores LR(k), $k > 1$

- utilizam os conjuntos $first_k$ e $follow_k$ para determinar o k -ésimo símbolo dos não terminais para os analisadores $SLR(k)$, $LALR(k)$ e $LR(k)$.
- estas generalizações aumentam as classes de gramáticas que podem ser processadas: $LR(0) \subset LR(1) \subset \dots \subset LR(k) \subset LR(k+1)$ onde $LR(k)$ designa o conjunto de gramáticas processáveis por uma técnica LR com k símbolos de antevisão.
- para cada técnica LR com k símbolos de antevisão temos ainda que: $LR(k-1) \subset SLR(k) \subset LALR(k) \subset LR(k) \subset SLR(k+1)$.
- contudo a dimensão das tabelas cresce rapidamente com k .
- as gramáticas que exigem $k > 1$ podem ser transformadas em $SLR(1)$.
- todas as linguagens analisáveis deterministicamente têm gramáticas $SLR(1)$.

Analísadores ascendentes LR

- nem todas as gramáticas livres de contexto são LR.
- uma gramática ambígua nunca é LR.
- como o número de símbolos de antevisão é fixo, apenas linguagens e gramáticas determinísticas podem ser tratadas.
- contudo, um analisador LR é:
 - é eficiente: proporcional à dimensão da entrada.
 - é correto: deteta todas as entradas não válidas e produz resultados corretos para entradas válidas.
 - é não ambíguo: produz resultados deterministas.
 - é linear no espaço e no tempo face ao número de *tokens* de entrada.

Árvore sintática

- Uma árvore sintática é uma representação sintática, mas a sua construção é, em geral, considerada semântica pois é efetuada por ações semânticas.
- Cada nó da árvore, modelado como uma estrutura com vários campos, designa um operador ou operando da linguagem.
- Cada operador é identificado por um atributo no nó da árvore, incluindo este nó ponteiros para os seus operandos.
- Os nós podem conter espaço para conter informação adicional sobre o operador e o seu contexto de utilização.

Construção da árvore sintática

- A árvore é constituída por valores literais nas folhas e operadores nos nós, todos designados pela estrutura *Node*.
- Cada *Node* permite associar um atributo, que descreve a funcionalidade, e um ponteiro para uso genérico (*user*).
- Na impressão pode ser fornecida uma tabela de cadeias de caracteres contendo a designação dos atributos. Por exemplo, onde *tab[OPadd] == "add"*.
- O tipo e o atributo de cada nó podem ser alterados independentemente, desde que o tipo coincida com os valores armazenados, permitindo substituir:
 - um argumento formal de uma rotina pela sua posição.
 - uma cadeia de caracteres pelo número do *label* que a identifica.

Construção da árvore sintática

- Construção dos nós da árvore:
 - sem valor:** *Node * nilNode(int attrib)*
 - inteiro:** *Node * intNode(int attrib, int value)*
 - real:** *Node * realNode(int attrib, double value)*
 - string:** *Node * strNode(int attrib, char * value)*
 - dados opacos:** *Node * dataNode(int attrib, int size, void * data)*
 - operadores:** *Node * subNode(int attrib, int nops, ...)*
*Node * addNode(Node * base, Node * node, unsigned pos);*
- Funções auxiliares
 - dados auxiliares:** *void * userNode(Node * p, void * user)*
 - libertar recursivamente o sub-ramo:** *void freeNode(Node * p)*
 - imprimir a árvore:** *void printNode(Node * p, FILE * fp, char * tab[])*

Exemplo de construção da árvore sintática

```
...
%union { int i; char *s; Node *n; }
%token<i> INT
%token<s> ID STRING
%type<n> expr decls
...
%%
file: decls { printNode($1, 0); freeNode($1); }
...
expr: expr '+' expr { $$ = subNode(OPadd, 2, $1, $3); }
    | '-' expr      { $$ = subNode(OPminus, 1, $2); }
    | INTEGER       { $$ = intNode(OPint, $1); }
    | STRING        { $$ = strNode(OPstr, $1); }
    | ID            { $$ = strNode(OPid, $1); }
...
```

Manipulação de nomes

- **visibilidade:** capacidade em aceder ao valor de uma variável através do seu nome:
 - estática ou léxica: determinável em tempo de compilação.
 - dinâmica: determinável em tempo de execução.
- **alcance:** capacidade em aceder ao valor de uma variável.
- **tempo de vida:** tempo que media entre a criação da variável e a sua destruição:
 - global: desde o início da execução até ao fim.
 - local: desde o início do bloco/função até ao fim.
 - *heap*: criado pelo utilizador e libertado por este ou por *garbage collector*.

Registo de alcance

Registo de alcance: permite a determinação das características de um símbolo:

- nome do símbolo
- classe do símbolo: label, função, variável, array, ...
- tipo do símbolo: inteiro, float, double, ...
- informação específica do tipo: limites do array, número e tipo dos argumentos de uma função, ...
- endereço onde o símbolo reside, se já estiver definido.
- lista de locais onde foi referido e ainda não foi colocado o endereço.

Tabela de símbolos

- **Tabela de símbolos:** estrutura que permite determinar, em cada ponto do programa, os símbolos visíveis:
 - **por alcance:** uma árvore onde os nós incluem os símbolos com o mesmo alcance (corresponde aos blocos da maioria das linguagens de programação).
 - **global:** uma lista única onde cada registo inclui informação sobre o início e fim do alcance de um símbolo.
- tipos de tabelas de símbolos: tabela, lista, árvore ou tabela de dispersão (*hash table*).
- **forward declaration:** torna visível um símbolo (variável/função) que está no alcance da declaração. (permite o acesso a nomes não locais)

tabid: uma tabela de símbolos linear

Definido nos ficheiros `tabid.h` e `tabid.c`:

- **int IDnew(int tipo, char *símbolo, int atributos):** introduz um novo símbolo no nível corrente com o tipo e atributos indicados. Devolve **1** se já existir neste nível e **0** caso contrário.
- **void IDpush(void):** cria um novo nível (bloco).
- **void IDpop(void):** retira o último nível.
- **int IDfind(char *símbolo, int *atributo):** procura um símbolo desde o nível atual até ao primeiro nível (global). Devolve **0** caso o símbolo não seja encontrado ou o tipo passado em `IDnew`. Caso o último argumento seja não nulo é preenchido com os atributos do símbolo.
- **void *IDroot(void *swap):** troca a origem da tabela, devolvendo a antiga raiz.

Tipos de dados

- tipo de dados: é um conjunto de entidades caracterizadas por uma representação estrutural e um conjunto de operações que lhe conferem um comportamento comum.
- tipo de dados abstrato: definido apenas pelas operações que oferece.
- consistência de tipos: regras que determinam se uma operação pode atuar sobre um tipo.
 - fraca: o sistema de tipos adapta implicitamente o tipo existente ao tipo necessário.
 - forte: os tipos devem ser compatíveis (podem ser sub-tipos). A conversão deve ser efetuada pelo programador através de funções ou *casts*.
- equivalência de tipos: quando dois tipos são considerados o mesmo tipo (estrutural ou por nome).

Tipos das linguagens

- **tipos básicos:** tipos de dados definidos na maioria dos processadores:
 - carateres:** sem operações especiais além de cópia.
 - inteiros:** aritmética básica e eficiente.
 - vírgula flutuante:** operações matemáticas complexas e lentas.
- **tipos construídos:** derivados dos tipos básicos com suporte no processador:
 - ponteiros:** acesso indireto a posições de memória (um inteiro com uma interpretação diferente).
 - vetores:** operações de referência e desreferência.
 - funções:** sequências de caracteres com instruções e dados imediatos do processador.
- **tipos compostos:** classes, estruturas, enumerados e uniões.

Tipificação

- tipificação: ato de verificação de consistência de tipos:
 - estática: executada no momento da compilação.
 - dinâmica: executada durante a execução, no momento do acesso. (*message not understood* em Smalltalk)
- subtipo: um tipo diz-se subtipo de outro tipo se realizar todas as suas operações com a mesma semântica, podendo acrescentar estrutura e operações.
- sub-tipificação: definição de um tipo a partir de outro:
 - explícita: por extensão através da especialização explícita de um tipo existente; ajuda a manter a semântica mas não a garante (herança).
 - implícita: relacionamento entre dois tipos quaisquer onde um tipo é subtipo de outro (mais geral e flexível, mas mais perigoso).

Polimorfismo

Polimorfismo: os valores, variáveis e funções podem ter mais de um tipo simultaneamente. Reduz a complexidade (menos nomes de funções) e melhora a abstracção (o mesmo nome é usado para a mesma operação em entidades distintas).

- sobreposição (*overloading*): escolha do método que realiza a operação depende do número e tipo de argumentos (multi-métodos).
- coerção: conversão de tipos explícita (*casts*) ou implícita (tipificação fraca).
- inclusão (*overriding*): escolha do método que realiza a operação depende do tipo de objeto criado e não do tipo declarado. Apesar da tipificação estática o *binding* não pode ser estático.
- paramétrico (ou genericidade): o tipo é determinado por um parâmetro atual na instanciação. A declaração é construída à custa de um tipo formal (abstrato).

Binding (*resolução de nomes*)

Binding é determinação da realização a utilizar por um símbolo:

- Quando o programa é escrito
- Quando o programa é compilado
- Quando o programa é editado, mas antes do seu carregamento (biblioteca estática)
- Quando o programa é carregado (biblioteca dinâmica)
- Quando um registo de base, usado para endereçamento, é carregado (Object-oriented)
- Quando a instrução contendo o endereço é executada.

Verificações estáticas

Tudo o que poder ser verificado em tempo de compilação não deve ser deixado para a execução:

- Verificação de tipos: determinação da compatibilidade entre dados e operações.
- Verificações de fluxo de controle: `break` dentro de `switch` e ciclos, *etc.*
- Verificações de existência e unicidade: de identificadores e `labels`.
- Verificações de contexto: *tags* no início e fim dos blocos em **Ada** ou **Visual basic**.

Árvores de ativação

- A definição de uma rotina é constituída por um nome, um conjunto de `parâmetros formais` e um corpo.
- Uma ativação de uma rotina, ou chamada à rotina, é constituída por um nome e um conjunto de `parâmetros atuais`.
- A árvore de ativação é o conjunto de ativações em uso num determinado instante.
- Uma rotina recursiva permite criar mais de uma ativação dessa rotina na árvore. (a recursão pode ser indireta)
- Se duas ativações, de rotinas distintas, não se sobrepõem na árvore então as rotinas não são aninhadas.
- a pilha, quando existe, é a forma mais usada para construir a árvore de ativação das rotinas.

Registo de ativação (*stack frame*)

- Valores dos parâmetros atuais. Podem ser endereços (passagem por referência), nomes (passagem por nome), *etc.*
- Contador do número de argumentos, se for desconhecido no momento da compilação
- Endereço de retorno à rotina chamadora.
- Valores de retorno (em **C** as rotinas retornam no máximo um só valor e o retorno é feito no acumulador) e sua localização (registos, pilha ou memória).
- Endereço do registo de ativação anterior (*frame pointer*)
- Variáveis locais da rotina chamada.

Compilação separada implica que a construção do registo de ativação é feito em colaboração pelas duas rotinas.

Passagem de argumentos

- **valor:** o valor (*rvalue*) do parâmetro é colocado no registo de ativação. As modificações efetuadas na rotina chamada não afetam o valor original.
- **referência:** o endereço (*lvalue*) do parâmetro é colocado no registo de ativação. As modificações efetuadas na rotina chamada afetam diretamente o valor original.
- **copy-restore:** passagem por valor em que o resultado é copiado de volta para a variável após o retorno da função. Idêntico à passagem por referência, mas imune aos efeitos laterais (*side effects*).
- **call-by-name:** a rotina comporta-se como uma macro sendo expandido no local da chamada. (`#define` do C)
- **call-by-need:** idêntico ao **call-by-name** mas preservando o compartilhamento de avaliações, ou seja, evita avaliações repetidas. (`inline` do C++)

Convenções de chamada

- Existem diversas convenções de chamada definidas por linguagens, compiladores e sistemas operativos (para *system calls*), mas as mais comuns são:
- **Cdecl:** argumentos são avaliados da esquerda para a direita, mas colocados na pilha da direita para a esquerda (primeiro argumento no topo). A rotina chamadora é responsável por retirar os argumentos da pilha e o valor de retorno é devolvido no acumulador.
- **Pdecl:** mesma ordem de avaliação, mas os argumentos são colocados na pilha da esquerda para a direita. A rotina chamada é responsável por retirar os argumentos da pilha e colocar o resultado no acumulador.
- **STDecl:** igual ao **Cdecl** mas quem retira os argumentos é a rotina chamada.

Reserva de memória

- **estática:** com tempo de vida igual à duração da execução do programa podem ser iniciadas (`.data`), não iniciadas (`.bss`), constantes (`.text`),
- **pilha:** com tempo de vida igual à duração da execução da rotina podem ser argumentos, variáveis locais e temporárias, e reserva dinâmica (ver adiante).
- **heap:** o tempo de vida é controlado pelo programador através de chamadas específicas (`new/delete`), podendo a libertação ser controlada por um *garbage collector*.
- a reserva dinâmica na pilha (`alloca` em C) permite criar variáveis de comprimento determinado em tempo de execução, não redimensionáveis (o custo de libertação é nulo).
- endereços de entidades criadas na pilha não podem ser devolvidos da rotina que criou essas entidades (*dangling pointers*).

Síntese

- Tradução do programa analisado para um formato de destino, retendo a semântica original da linguagem.
- Formas mais comuns de síntese:
 - interpretação.
 - tradução dirigida pela sintaxe.
 - tradução dirigida pela árvore sintática.
 - tradução dirigida por DAG.
- A síntese é, em geral, realizada em três passos:
 - geração de código intermédio (processador simplificado imaginário).
 - geração de código final (para um processador específico).
 - otimização de código (no código intermédio e final).

Interpretação

- Execução de um programa por outro programa (o interpretador), em vez do processador da máquina utilizada.
- Interpretação linha a linha: só para gramáticas muito triviais.
`while(gets(buf)) execute(buf);`
- Interpretação dirigida pela sintaxe: só para linguagens muito simples (sem ifs ou **whiles**).
- Interpretação dirigida pela árvore sintática: muito lento.
- Interpretação por máquina virtual (*Threading*): o mais rápido e usado comercialmente.
- Interpretação de processadores reais (emulação): complexo e lento, quando efetuado exclusivamente por *software*.

Interpretação dirigida pela sintaxe

- Utiliza os dados disponíveis na pilha de dados do analisador sintático:

```

expr: INTEGER      { $$ = $1; }
    | IDENTIFIER   { $$ = getValue($1); }
    | IDENTIFIER '=' expr { setValue($1, $$ = $3); }
    | expr '+' expr { $$ = $1 + $3; }
    | '-' expr { $$ = - $2; }

```

- Aproveita a pilha de dados do analisador sintático para transportar os valores intermédios da interpretação das operações.

Interpretação dirigida pela árvore sintática

- Constrói a árvore dirigida pela sintaxe e depois interpreta a árvore gerada.
- Interpretação da árvore:

```

int eval(Node *n) { ...
    switch(n->type) { ...
        case OPinteger: return n->value.i;
        case OPident: return getValue(n->value.s);
        case OPassign: {
            int ret = eval(n->value.sub.n[1]);
            setValue(n->value.sub.n[0].value.s, ret);
            return ret; }
        case OPadd: return eval(n->value.sub.n[0]) +
            eval(n->value.sub.n[1]);
        case OPuminus: return -eval(n->value.sub.n[0]);
    }
}

```

Construção da árvore sintática

- A árvore sintática é construída usando as acções da tradução dirigida pela sintaxe:

```

expr: INTEGER      { $$ = intNode(OPinteger, $1); }
    | IDENTIFIER   { $$ = strNode(OPident, $1); }
    | IDENTIFIER '=' expr { $$ = subNode(OPassign,
        2, strNode(OPident, $1), $3); }
    | expr '+' expr { $$ = subNode(OPadd, 2, $1, $3); }
    | '-' expr { $$ = subNode(OPuminus, 1, $2); }

```

Interpretação por máquina virtual (*Threading*)

- As operações a executar estão codificadas sob a forma de endereços simbólicos, ou sob a forma de códigos (*bytecodes*) que representam deslocamentos numa tabela que contém esses endereços.
- O interpretador executa cada uma dessas operações em sequência:
 - routine threading:** cada operação é codificada numa rotina.
 - case threading:** cada operação é uma opção da instrução *switch* do interpretador (só para *bytecodes*).
 - direct threading:** cada operação é identificada por um *label* e, no fim da qual, existe um salto direto para a operação seguinte.
 - indirect threading:** idêntica à anterior, mas o salto é indireto (esta é forma mais vulgar de *threading*).

Notação postfixada: Postfix

- Notação em que os operandos precedem os operadores (também conhecida como *reverse polish notation*). Por exemplo, $(2 + 3) * 4 + 5$ passa a ficar $2\ 3 + 4 * 5 +$ ou $5\ 4\ 2\ 3 + * +$
- Muito utilizada em como formato intermédio de linguagens, por exemplo, PostScript, Forth, Java, Smalltalk e .net.
- Facilidade na reconstrução da árvore sintática a partir do código, para compiladores JIT.
- Vantagem: grande facilidade de geração, quer a partir da árvore sintática, quer dirigida pela sintaxe, podendo ser utilizada como código final (*stack machines*) ou código intermédio. O código gerado é muito compacto (ocupa pouco espaço).
- Desvantagem: sequencialidade explícita, o que dificulta a paralelização e otimização.

Postfix: processador imaginário SL0

- SL0: single stack, large stack size, zero operand machine.
- todos os valores locais são guardados na pilha de dados. Os restantes valores são globais.
- a pilha tem dimensão ilimitada: não gera *overflows* (exceto quando não existe mais memória utilizável).
- 3 registos: **ip** (*instruction pointer*), **sp** (*stack pointer*) e **fp** (*frame pointer*).
- os operandos estão no topo na pilha, sendo substituídos pelo resultado da operação.

Postfix: tradução dirigida pela sintaxe

- Quer os operandos quer o operador estão na pilha do analisador sintático:

```

expr: LITERAL      { emit($1.value); }
    | IDENTIFIER    { emit($1.name); }
    | expr '+' expr { emit("add"); }
    | '-' expr      { emit("uminus"); }
    | expr '?' jz expr ':' jmp { emit(".L%d:", $3); }
                                expr { emit(".L%d:", $6); }
    ;
jz  : { emit("jz .L%d", ++lbl); $$ = lbl; } ;
jmp: { emit("jmp .L%d", ++lbl); $$ = lbl; } ;

```

- Vantagem: aparente simplicidade na geração de código.
- Desvantagem: introduz ações e regras nulas que podem produzir conflitos (em parsers LR(*k*) com $k \leq 1$).

Postfix: tradução por árvore sintática

- Cada nó da árvore designa um operador e os seus ramos os operandos:

```
void code(Node *t) { ...
  switch (n->type) { ...
    case operador-N: /* operador com M argumentos */
      code(n->value.sub.n[0]);
      ...
      code(n->value.sub.n[M]);
      emit("operador-N");
      break;
```

- Vantagem: modularidade (separa a síntese da análise), facilidade na geração de código. Passo intermédio para a transformação em DAG (otimização).
- Desvantagem: obriga à construção da árvore sintática (necessita de memória).

tradução por árvore sintática: execução condicionada

- geração de código que implica saltos:

if	expr	then	instr	end
	(expr)	jz L1	(instr)	L1:

if	expr	then	instr1	else	instr2	end
	(expr)	jz L1	(instr1)	jmp L2; L1:	(instr2)	L2:

while	expr	do	instr	done
L1:	(expr)	jz L2	(instr)	jmp L1; L2:

expr1	and	expr2	
(expr1)	dup; jz L1; trash	(expr2)	L1:

expr1	or	expr2	
(expr1)	dup; jnz L1; trash	(expr2)	L1:

Diretivas de assembler

global : torna símbolos locais acessíveis por outros módulos.

extern : torna acessíveis símbolos de outros módulos.

common : torna símbolos comuns (partilhados) entre módulos.

align : alinha a memória antes de cada variável, constante ou função.

segment : altera o segmento corrente

.data : dados iniciados por **db** (*define byte*), **dd** (*define word*, 4 bytes).

.bss : dados não iniciados por **resb** (*reserve byte*).

.text : funções (*read only*).

.rodata : constantes e strings literais (*read only*).

Em *coff* e *win32* designa-se por **.rdata** e não existe em *aout*.

Questões práticas

- Operadores retiram argumentos da pilha e colocam o resultado na pilha.
- Constantes escalares são empurradas na pilha enquanto variáveis são colocadas na área de dados. Constantes vetoriais, incluindo strings, devem ser colocados na área de texto pois não podem ser modificados (*read only*).
- Um *lvalue* é um endereço no qual pode ser efetuado um *store*. Um *lvalue* pode ser convertido num *rvalue* desreferenciando o endereço com um *load*.
- a sequência de chamada a uma rotina inicia-se colocando os argumentos, com o primeiro argumento no topo, seguido da chamada e, após o retorno, retirar os argumentos da pilha.
- uma rotina ao ser chamada cria o novo registo de ativação (*stack frame*) e reservado espaço para as variáveis locais; no fim, e antes do retorno, deve ser retirado o registo de ativação.

Seleção de instruções

- A seleção de instruções pode ser crítica em processadores com muitas instruções, que permite realizar a mesma operação recorrendo a diferentes combinações de instruções (processadores CISC, tipicamente).
- Em processadores com poucas instruções (processadores RISC ou stack-machines) as escolhas são poucas, ou nenhuma, pelo que a seleção de instruções não é crítica.
- Cada instrução do processador é representada como uma árvore padrão e por um custo associado. O custo é definido arbitrariamente e pode ser utilizado para minimizar o tempo de execução ou o espaço ocupado, por exemplo.
- Existem dois tipos de algoritmos para a seleção de instruções: maximal-munch e programação dinâmica.

Seleção de instruções por maximal-munch

- A seleção de instruções por maximal-munch -e um algoritmo *greedy* que consiste numa só passagem descendente (da raiz para as folhas) de uma árvore sintática.
- Em cada passo é escolhida a instrução com o padrão mais extenso que se adapta ao nó em análise, e a respetivo código é gerado.
- As instruções são geradas por ordem inversa.
- Como só considera o nó em análise e não os sub-nós a solução não é globalmente ótima, apenas localmente ótima.
- O algoritmo é muito eficiente e produz código de boa qualidade em arquiteturas RISC (instruções de pequena profundidade).

Seleção de instruções por programação dinâmica

- Uma seleção ótima necessita de dois passos: primeiro calculam-se todos os custos e no segundo gera-se as instruções de menor custo.
- Cálculo dos custos é efetuado das folhas para a raiz (*bottom-up*), em cada nó considera-se a melhor instrução local, à qual se soma os menores custos de cada sub-ramo. Notar que ao subir na árvore podem-se ir encontrando instruções mais complexas e de menor custo que a seleção local.
- A escolha das instruções é efetuada da raiz para as folhas, como no maximal-munch, mas tendo em conta os custos ótimos em cada nó.
- O algoritmo é indicado para processadores com grande combinatória de instruções, mas é mais lento pois exige dois passos e, para processadores mais complexos, a árvore é maior.

Gramáticas burg

O formato das regras é constituído por uma etiqueta seguido da instrução; depois da atribuição indica-se o número da regra (para identificar o código a gerar e, entre parentesis, o custo associado à instrução a gerar.

```
stat: mem = 0 (0);
mem: store(reg) = 1 (19);
mem: store(cte) = 2 (20);
reg: load(mem) = 3 (18);
reg: load(cte) = 4 (4);
reg: add(reg,reg) = 5 (3);
reg: add(reg,cte) = 6 (4);
reg: add(reg,mem) = 7 (19);
mem: add(mem,cte) = 8 (31);
mem: add(mem,reg) = 9 (30);
reg: uminus(reg) = 10 (3);
mem: uminus(mem) = 11 (30);
```

Cálculos dos custos (*labeling*)

- Construir os pares $\langle \text{ast-node}, \text{op-tree} \rangle$, garantindo que os ramos do nó intermédio concordam com os não-terminais dos nós filhos.

- Usar apenas 1 operação por *ast*:

$x : \text{op1}(a, \text{op2}(b, c)) \rightarrow x : \text{op1}(a, a') \wedge a' : \text{op2}(b, c)$

Tile(n)

Label(n) <- 0

Tile(child[0]) ... Tile(child[k])

for each rule that matches n

if child[r] in Label(child[n]) and ...

Label(n) <- Label(n) U {r}

- Utilização de custos dinâmicos obriga a fazer *labeling* no gerador. Pode ser necessário devido a *spilling* e *scheduling* dos dados (para evitar latência).

gerador Burg e derivados

Burg: Recebe uma árvore com a representação intermédia e calcula os custos quando o gerador é produzido, logo os custos são estáticos.

Iburg: ao contrário do **Burg**, calcula os custos quando o gerador executa, logo os custos podem ser dinâmicos.

Gburg: utiliza uma expressões regulares, logo produz uma máquina de estados finita (FSM), gerando as instruções num só passo. Tal estrutura adapta-se às representações de bytcodes postfixados. Utiliza um algoritmo *greedy* e produz código ao dobro da velocidade do **Iburg**

pburg: semelhante ao **Iburg**, mas com suporte para custos dinâmicos, blocos de código, acesso ao `y.tab.h`, etc.

Escalonamento de instruções

- Uma multiplicação pode ter uma grande latência, ou seja, leva muito tempo a produzir o resultado a partir dos dados. No entanto, o resto dos registos está livre e podem ser efetuadas outras operações (somas, por exemplo) enquanto o resultado da multiplicação não estabiliza.
- Escalonamento de instruções consiste em alterar a ordem de execução das instruções por forma a tirar proveito das limitações de desempenho do processador.
- Por exemplo, assumindo $\text{mul} = 2$, $\text{store} = 3$, $\text{add} = 1$ e $\text{load} = 3$, a expressão $w = w \times 2 \times x \times y \times z$ pode gastar 22 ciclos (*load, add, load, mul, load, mul, load, mul, store*) ou 13 ciclos (*load, load, load, add, mul, load, mul, mul, store*), como se pode verificar pelos tempos de início e fim de cada instrução.

Grafo de dependência e instruções acessíveis

- Renomear as definições para evitar antidependências: cada nova definição de uma variável recebe um novo nome (SSA ou C3E, por exemplo).
- Construir o grafo, percorrendo sequencialmente as instruções pela ordem inicial. Cada operação representa constitui um novo nó que representa o novo valor calculado. Os ramos, dirigidos, indicam os valores utilizados na operação e são etiquetados com a latência da operação corrente.
- Atribuir prioridades a cada operação, para determinar quais as instruções a selecionar em cada passo do escalonamento. A ordenação mais comum utiliza o caminho cuja soma das latências é maior como primeira escolha.
- Manter uma lista de instruções acessíveis (*ready*), em cada momento. Estão acessíveis todas as instruções cujos operandos estejam disponíveis.

Escalonamento da instruções por lista

- O escalonamento é efetuado por simulação abstrata:

```
t <- 0; Ativas <- {}, Ready <- Graph leaves
while (Ready U Ativas <> {})
  if (Ready <> {})
    remove Op from Ready and add to Ative
    Start(Op) <- t
  t <- t + 1
  for each Op in Ativas
    if (Start(Op) + Delay(Op) < t)
      remove Op from Ative
for each sucessor S of Op in Graph
  if (S is ready)
    add S to Ready
```

Utilização de registos genéricos

- Os registos disponibilizados pelo processador sem função específica designam-se por genéricos, excetua-se IP, SP, FP e flags. Não confundir com a classe do registo, ou seja, o tipo de operações a que pode ser sujeito, por exemplo, registo de base em operações de indexação ou registo contador.
- Existem processadores sem registos genéricos (*stack-machines*), com apenas um registo (*accumulator-machines*), com poucos (< 16) registos (*CISC-machines*) e muitos (≥ 16) registos (*RISC-machines*).
- O registos genéricos podem ser usados para:
 - resultados intermédios da avaliação de expressões
 - reutilização de resultados de expressões
 - conteúdo de variáveis frequentemente utilizadas
 - parâmetros ou resultados de funções,

Atribuição de registos

- A atribuição de registos é efetuada depois do código fonte ter sido analisado, otimizado, traduzido para código máquina e, por vezes, escalonado.
- Os registos são a memória de acesso mais rápido, mas o número de registos é limitado, em cada classe: endereço, inteiro, vírgula flutuante, ...
- A atribuição de registos procura reduzir:
 - número de acessos a memória: acessos a *cache* são $2\times$ mais lentos, a memória são $2\times$ a $10\times$ mais lentos e a disco (*swapping*) $100\times$ a $10000\times$ mais lentos.
 - *spilling* (ou rejeição): instruções que salvaguardam e repõem o conteúdo de registos, em geral por falta de registos disponíveis.
- O processo baseia-se em criar um novo registo simbólico para cada valor calculado, ficando a atribuição efetiva de registos para uma fase posterior.

Métodos de atribuição de registos

- A atribuição de registos deve atribuir registos físicos às instruções já geradas e introduzir instruções que movimentem os valores entre registos (no caso de registos não uniformes) e entre registos e memória.
- Se o número de registos não for suficiente acrescenta-se código (*spill code*) para guardar e repor o valor de registos em uso. O objetivo consiste em minimizar o *spill code* necessário através da alteração da ordem de avaliação.
- A atribuição de registos pode ser precedida por uma fase de reserva, onde os registos de cada instrução são pré-escolhidos e os *spills* necessários gerados.
- Existem diversos métodos de atribuição de registos:
 - local: *top-down*, ordenação de subárvores e próximo uso.
 - global (para além dos blocos básicos): *greedy*, *linear-scan*, *next-use* ou coloração de grafos.

Spilling

- o DAG minimiza a reavaliação de subexpressões comuns;
- o `spill` é, em grande medida, o resultado da *ganância* em evitar as reavaliações.
- o nó que representa a subexpressão comum é modificado para guardar o valor obtido, de tal forma que posteriores referências obtenham esse valor.
- *spilling* envolve 3 passos:
 1. identificar os registos a libertar (spilled registers)
 2. gerar o código que salvaguarda o valor do registo
 3. gerar, no local apropriado, o código que recarrega o valor salvaguardado.

Atribuição de registos *top-down*

- Heurística: os valores mais utilizados são mantidos em registos. Para tal conta o número de ocorrências de cada registo virtual no bloco básico, usando a respetiva frequência para definir as prioridades.
- Se houver mais registos virtuais que físicos, devem ser deixados 2 a 4 registos livres para carregar 2 operandos e guardar o resultado.
- Percorrer a lista linear de instruções, contando as ocorrências de cada registo virtual, e ordenar por ordem decrescente do número de ocorrências. Atribuir os k registos mais utilizados.
- Numa segunda fase, percorrer a lista de instruções atribuindo registos temporários aos registos ainda não atribuídos, introduzindo as instruções de `load` e `store` necessárias.
- Problema: registos só usados no início do bloco não são libertados.

Ordenação de subárvores (*Sethi & Ullman*)

- Minimizar o número de registos necessários começando por avaliar a subárvore que necessita de maior número de registos.
- A primeira passagem determina o número de registos necessários (*labeling*):

trivial		0
básico		1
unário		$\max(1, \text{need}[\text{child}])$
binário	$\text{need}[\text{left}] == \text{need}[\text{right}]$ $\text{need}[\text{left}] != \text{need}[\text{right}]$	$1 + \text{need}[\text{left}]$ $\max(1, \text{need}[\text{left}], \text{need}[\text{right}])$

- Na segunda passagem avaliar a subárvore que necessita de maior número de registos, efetuando a atribuição de registos (*top-down*) e gerar o código pela ordem de avaliação (*bottom-up*).
- O código gerado só é ótimo se os registos forem uniformes, mas não otimiza subexpressões comuns.

Atribuição de registos *greedy*

- Ocupar os registos sequencialmente, registando a data da ocupação (número de sequência da instrução, por exemplo).
- Quando não existem registos disponíveis, retirar o registo ocupado à mais tempo. (Claro que pode ser, precisamente, aquele que vai ser utilizado na instrução seguinte!)
- Pode ser melhorado mantendo a informação dos registos `limpos` (registos que foram carregado de memória e ainda não foram modificados) e `sujos` (cujo valor foi alterado pelo processador e ainda não foi guardado). Optar pelo registo `limpos` ocupado à mais tempo para evitar o *spilling*. Como pode nem ser necessário o `reload`, a reutilizam de um registo `limpos` pode ter custo zero.

Atribuição linear de registos

- A atribuição linear de registos usa código já linearizado (sem considerar blocos básicos) e constroi uma lista ordenada com o intervalo de vida de cada uma das variáveis.
- Cada instrução intermédia (linearizada) é atribuído um número com base no qual são definidos os limites do intervalo.
- Numa aproximação conservadora podem existir subintervalos em que a variável não está ativa. Um outro algoritmo, designado por *binpacking*, controla igualmente os subintervalos (*lifetime holes*) mas consome muito mais tempo.
- A atribuição de registos é efetuada de uma forma sequencial e são efetuados *spilling* sempre que não existem registos suficientes.

Algoritmo de Linear Scan

- Construir uma lista de intervalos, representando cada um o tempo de vida de cada variável, ordenada por ordem crescente de primeira utilização.
- Percorrer a lista e atribuir variáveis a registos, enquanto houver registos disponíveis. Se não houver registos fazer *spilling*.
- Manter uma outra lista de variáveis ativas, ordenadas por ordem crescente de fim de vida. Antes de cada nova atribuição, retirar as variáveis que entretanto expiraram.
- Para fazer *spilling*, considerar a última variável ativa: aquela cujo fim de vida é mais distante. Se o seu fim de vida for inferior ao da nova variável, manter a atribuição existente (fazer *spilling* da nova variável), caso contrário fazer *spilling* do registo e ocupá-lo com a nova variável.

Atribuição de registos por *próximo uso*

- Atribuição de registos com base numa tabela de símbolos de variáveis vivas e indicação de próximo uso. Uma variável está viva se voltar a ser usada.
- Começar no fim do bloco com todas as variáveis do programador vivas e todas as variáveis temporárias mortas. Nenhuma variável tem o próximo uso preenchido.
- Recuar até ao início do bloco, executando em cada instrução:
 1. Retirar da tabela de símbolos a informação sobre cada variável utilizada na instrução, morta ou viva e respetivo próximo uso (se viva) e associar à instrução;
 2. Atualizar a tabela de símbolos considerando: cada argumento como tendo a instrução corrente como próximo uso; a variável atribuída, caso exista, é marcada como morta e sem próximo uso.

Atribuição de registos por *próximo uso* (2)

- Exemplo: $d = (a - b) + (c - a) - (d + b) * (c + 1)$

a	V					a	1
b	V		C3E	vivas	mortas	b	1
c	V	1	$u = a - b$	u:3 a:2 b:4		c	2
d	V	2	$v = c - a$	v:3 c:5 a:-		d	4
u	M	3	$w = u + v$	w:7	u v	u	M
v	M	4	$x = d + b$	x:6 b:-	d	v	M
w	M	5	$y = c + 1$	y:6 c:-		w	M
x	M	6	$z = x * y$	z:7	x y	x	M
y	M	7	$d = w - z$	d:-	w z	y	M
z	M					z	M

Atribuição de registos por *próximo uso* (3)

- A tabela de símbolos em cada passo é:

passo	linha	C3E	a	b	c	d	u	v	w	x	y	z
			V	V	V	V	M	M	M	M	M	M
1	7	$d = w - z$				M			7			7
2	6	$z = x * y$								6	6	M
3	5	$y = c + 1$			5						M	
4	4	$x = d + b$		4		4				M		
5	3	$w = u + v$					3	3	M			
6	2	$v = c - a$	2		2			M				
7	1	$u = a - b$	1	1			M					

Atribuição de registos por *próximo uso* (4)

- usar o registo (R) se a variável já existe em (R) e (R) não é usado também por outra variável e a variável está morta (e sem próximo uso)
- se existir um registo livre, usá-lo
- selecionar um registo em uso com base numa tabela de registos (o que o registo contém) e numa tabela de endereços (onde existe o valor atual da variável: memória e/ou registo).
- instruções de atribuição, $a = b$, colocam duas variáveis no mesmo registo.
- variáveis que ficam sem *próximo uso* libertam o respetivo registo, antes de guardar o resultado.
- no fim do bloco, guardar as variáveis vivas.

Atribuição de registos por *próximo uso* (5)

- O exemplo, $d = (a - b) + (c - a) - (d + b) * (c + 1)$, pode ser codificado com 4 registos e sem *reloads*:

1	load a, r1	a:r1
	load b, r2	a:r1 b:r2
	sub r1, r2, r3	a:r1 b:r2 u:r3
2	load c, r4	a:r1 b:r2 u:r3 c:r4
	sub r4, r1, r1	v:r1 b:r2 u:r3 c:r4
3	add r3, r1, r1	w:r1 b:r2 c:r4
4	load d, r3	w:r1 b:r2 d:r3 c:r4
	add r3, r2, r2	w:r1 x:r2 c:r4
5	add r4, 1, r3	w:r1 x:r2 y:r3
6	mul r2, r3, r2	w:r1 z:r2
7	sub r1, r2, r1	d:r1
	store r1, d	

Atribuição de registos por coloração

- Construção de um grafo de interferências:
 - nós:** são *live ranges* (variáveis, temporários, registos virtuais/simbólicos) que são candidatos para a reserva de registos.
 - arcos:** ligam as *live ranges* que interferem, ou seja, que estão simultaneamente ativas pelo menos num ponto do programa. Assim, argumentos distintos de uma mesma operação correspondem a variáveis que não podem ser atribuídas ao mesmo registo.
- A coloração do grafo consiste atribuir cores, tantas cores quanto o número de registos disponíveis, aos nós de tal forma que nós que interferem possuem cores distintas (problema dos mapas políticos).

Impossibilidade de coloração

- Quando não existem cores suficientes é necessário remover alguns nós até que seja possível a coloração. Os nós removidos terão de ser *spilled*.
- O objetivo consiste em obter um grafo colorável em que o custo das operações de *spilling* seja mínimo. Isto não significa fazer um número mínimo de remoções de nós, já que certos nós provocam vários *spills*.
- Quando uma origem e o destino de uma operação não estão interligados nos grafo de interferências, os respetivos nós podem ser fundidos num só e a operação de cópia de registos eliminada. Contudo, esta técnica de fusão, designada por *coalescing*, quando usada em exagero pode conduzir a grafos não coloráveis e a mais *spilling*.

Otimização

Preservar o significado do programa original melhorando-o significativamente:

- geração de código ótimo é um problema indecidível
- heurísticas podem gerar código bom, mas não ótimo.
- deverá afetar significativamente o desempenho do programa.

Melhorar o resultado final do programa eliminando instruções:

inacessíveis: instruções que nunca conseguem ser executadas.

redundantes: instruções que não alteram o estado da aplicação.

lentas: conjuntos instruções equivalentes que executam mais depressa.

pesadas: conjuntos instruções equivalentes que ocupam menos memória e/ou registos do processador.

Tipos de otimização

utilizador: efetuadas pelo programador (ou aplicação) quando gera código fonte.

genérica: efetuada no código intermédio e independente do processador alvo.

peephole: considera apenas uma janela de instruções consecutivas.

local: otimização dentro de um bloco básico.

global: fluxo de informação entre blocos, essencialmente ciclos.

Blocos básicos

- Sequência de comandos consecutivos onde o fluxo de controlo começa na primeira instrução e termina na última, sem possibilidade de desvio exceto na última instrução:
 - um bloco básico começa numa etiqueta (ou etiquetas consecutivas) ou na instrução que se segue a um salto.
 - um bloco básico termina numa instrução de salto ou na instrução que antecede uma etiqueta.
- As transformações num bloco básico têm de preservar a semântica atômica do bloco:
 - eliminação de subexpressões comuns
 - troca da ordem de execução de instruções independentes
 - renomeação de variáveis

Subexpressões comuns num bloco básico

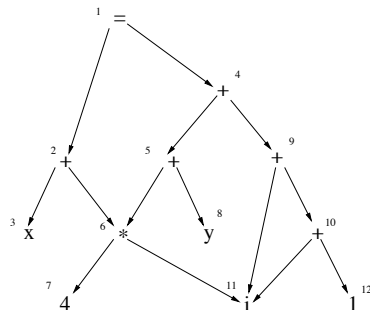
- Guardar referências para todos os nós criados.
- Procurar os nós que representam os argumentos de uma operação, dependendo da sua aridade. Se o nó não existir então deve ser criado.
- Se os nós de todos os argumentos já existiam, procurar um nó com a mesma operação que ligue os mesmos argumentos, ou seja, uma subexpressão comum.
- Ao gerar o grafo dirigido acíclico (DAG), considerar a atribuição simples (a := b) que acrescenta uma etiqueta ao nó, bem como o facto de qualquer atribuição a uma etiqueta mover uma anterior referência para o nó atual.
- O algoritmo pode ser realizado com recurso a uma pilha auxiliar para memorizar os argumentos, quer o código intermédio seja C3E ou postfixado.

Geração das instruções a partir de um DAG

- Num grafo, um nó constituído apenas por origens de arcos é designado por uma **fonte**, enquanto um nó constituído apenas por destinos de arcos é designado por **sorvedoro**.
- A ordenação topológica de um DAG baseia-se em numerar sucessivamente as **fontes**, por ordem crescente, retirando-as de seguida, até não haver mais nós.
- Se na escolha da **fonte** se optar pelo filho mais à esquerda da última fonte retirada, o código gerado é de qualidade superior pois valor calculado pela instrução fica diretamente acessível num registo.
- A geração de código (só C3E) é efetuada pela ordem inversa da numeração efetuada, ou seja, pode ser realizado com recurso a uma pilha auxiliar sem necessitar de numeração. (a lista de fontes do DAG pode ser também mantida numa pilha).

Exemplo de *selection sort*

- Considere-se a expressão: $x[i] = y[i] + i + 1$ representada pelo DAG abaixo, onde os nós estão etiquetados pela ordem inversa em que devem ser geradas as respetivas instruções:



Exemplo de *selection sort*

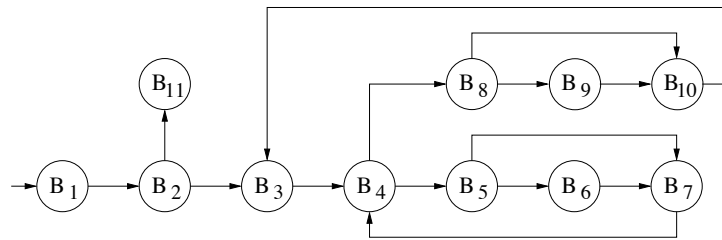
- Considere-se o algoritmo de ordenação:

```
void sort(int *vec, int siz)
{
    int i, j, max, tmp;

    for (i = siz; i > 1; i--) {
        max = 1;
        for (j = 2; j < i; j++) // o maior
            if (vec[j] > vec[max])
                max = j;
        if (max != i) { // colocar no lugar
            tmp = vec[i];
            vec[i] = vec[max];
            vec[max] = tmp;
        }
    }
}
```

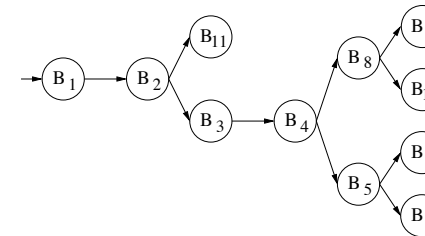
Grafo de fluxo de controlo

- Um grafo de fluxo de controlo representa as dependências entre blocos básicos:



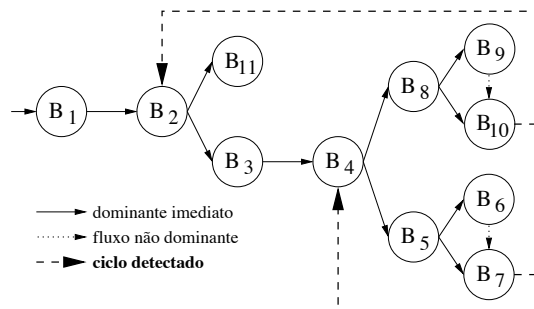
Cálculo de dominantes

- Um nó B_i **domina** um nó B_j se tivermos de passar por B_i para atingir B_j .
- B_i é um **dominante imediato** de B_j se B_i for o último **dominador** antes de atingir B_j .
- Os dominadores podem ser representados como um árvore, ou seja a dominação é uma ordenação parcial.



Deteção de ciclos

- se a origem de um arco é dominado pelo destino do arco então existe um ciclo. (ciclos são importantes em otimização pois representam a maioria do tempo de execução)



Melhorar o desempenho

predição de saltos (branch prediction): reduzir saltos, reduzir chamadas indiretas, *inlining*, *tail-recurse*, *prefetch* consistente com predição estática.

esperas de memória (memory access stalls): *store forwarding*, alinhamento, evitar código automodificável, evitar conversões de tipos, arrumação de estruturas, evitar variáveis globais, restringir alcance (static), constante em rodata, *prefetching*.

seleção de instruções: reduzir a latência (mul->shift->inc->add), evitar instruções complexas (micro-código).

sequenciação de instruções (instruction scheduling): calcular endereços o mais cedo possível, ter em conta a latência das instruções, evitar *load* → *store*.

vectorização: tipos de dados o mais pequenos possível, evitar condições, variável de ciclo com base em expressões simples, evitar *lexically-backward dependence*

Otimização pelo utilizador

Melhorar o programa pode degradar a sua legibilidade, manutenção e evolução:

zonas a otimizar: apenas as partes mais executadas (programas gastam 90% do tempo em 10% do código), usar sempre um *profiler*.

substituição de algoritmos: mais eficientes ou que ocupem menos espaço (*bubble* vs *quick sort*).

transformações de ordem elevada: aquelas que pela sua complexidade, ou falta de informação, dificilmente podem ser efetuadas por um compilador.

otimizações independentes da máquina: desde que não degradem significativamente a legibilidade do código, a menos que sejam críticas. (nem todos os compiladores têm optimizadores e muitos optimizadores são fracos).

Otimização independente da máquina

desdobramento e propagação de constantes (*constant folding*): resolução de expressões com literais durante a compilação ($MAX * 3 + 1$).

eliminação de código inacessível: após intruções de salto incondicional (*return*, *continue*, *break*, *etc.*), condições determináveis em compilação (*if (debug)*).

inline de funções: substituir chamadas a funções pelo código das próprias funções. (funções estáticas podem ser eliminadas e chamadas por valor obrigam a duplicar variáveis)

propagação de cópias: referências a uma variável podem ser substituídas por outra que lhe tenha sido atribuída (depois de $a = b$ existe um único valor que pode ser referido, até que uma das variáveis seja alterada).

ciclos: fusão e desdobramento de ciclos, deslocamento de código invariante, predição de saltos (ciclos *do*).

Otimização dependente da máquina

simplificações algébricas: instruções equivalentes, em geral com literais (por exemplo, $x + 0$, $x - x$, $x * 1$).

redução de esforço (*strength-reduce*): instruções equivalentes de menor custo em termos de desempenho ou memória (por exemplo, $x^2 = x * x$, $0 - x = -x$, $x * 2 = x << 1$).

peephole: simplificação de operações e utilização de instruções especiais com base na análise de uma janela de instruções (por exemplo, instruções **postfix** como *JEQ*, *ADDRV* ou *START*).

eliminação de instruções redundantes: que não afetam o estado do sistema: *store* seguido de um *load* da mesma posição de memória ou um *push* seguido de um *pop*.

Pentium IV face aos anteriores *

redução da latência das instruções: *add*, *sub*, *cmp*, *test*, *and*, *or*, *xor*, *neg*, *not*, *sahf*, *mov*.

aumento da latência das instruções: *shifts*, *rotates*, *mov* de memória com extensão de sinal.

separar: leituras e escritas da mesma posição de memória.

ciclos: devem terminar sempre com saltos para a frente (*branch prediction*).

chaches de 64 bytes: menor necessidade de *prefetching* (Pentium II e III usavam 32 bytes).

chamadas a procedimentos: saltos são mais dispendiosos que chamadas.

Outras otimizações *

Otimização independente da máquina:

ciclos: eliminação de variáveis de indução e simplificação de operações.

eliminação de subexpressões comuns: guardar numa variável temporária (registo ou memória) o resultado.

eliminação de código redundante: instruções sem efeito (push seguido de pop, store seguido de load).

Otimização dependente da máquina:

alteração da ordem das instruções: instruções com efeitos independentes podem ser alteradas por forma a reduzir o *spill* de registos.

troca de registos: evitar movimentos entre registos (escolha de registos).

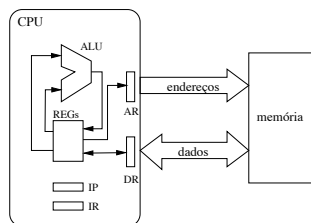
code hoisting: otimização do espaço ocupado e não da velocidade de execução.

Geração de código

- Produção de código para um processador específico, tendo em conta as suas características particulares.
- Tipos de processadores podem ser:
 - Stack machines: B5000, HP-3000 ou máquinas virtuais **Java**, e **.net**.
 - Accumulator machines: PDP-9, M6809, 68HC11.
 - Load-store machines (RISC): RS/6000, MIPS-R, HP-PA, SPARC, M88k.
 - Memory machines (CISC):
 - * Register-memory architecture: i80x86, IBM-RT, M68k, IBM360.
 - * Memory-memory architecture: VAX, PDP-11.
- Compilação JIT a partir de um formato intermédio ou compilação *object-code* a partir de código final.

Load-store machines

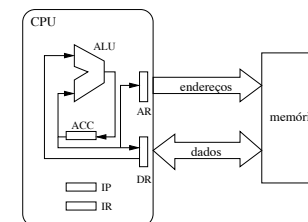
- **Load-store machines:** vários registos genéricos, todos os (3) argumentos explícitos:



- os argumentos de têm origem num registo bem como o destino do resultado. (operações especiais de *load* e *store* são necessárias)
- o resultado pode ser implícito num dos argumentos (primeiro ou segundo) ou num registo específico (acumulador).

Accumulator machines

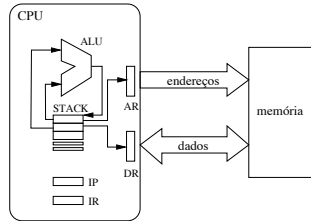
- **Accumulator machines:** só um registo (o acumulador), um argumento explícito:



- um dos argumentos tem origem no acumulador que serve também de destino para o resultado (o outro argumento vem da memória).

Stack machines

- **Stack machines:** sem registos, sem argumentos explícitos (0 address).



- argumentos têm origem na pilha bem como o destino do resultado.
- apenas a operação *literal* tem um argumento explícito.
- *load* e *store* são efetuados para e da pilha.

Memory machines

- Memory machines: máquinas híbridas onde o acesso à memória pode substituir alguns dos argumentos:
 - **register-memory:** um dos argumentos e/ou o resultado podem ser acedidos diretamente na memória, sem passar por um registo.
 - **memory-memory:** os dois argumentos, e mesmo o resultado, podem ser acedidos diretamente na memória, sem passar por um registo.
- estas máquinas só funcionam aceitavelmente com grandes caches (primária e secundária), pois o acesso à memória de massa é lento.

Operação de soma de inteiros

- Somar dois inteiros localizados em 100 e 104 e colocar o resultado em 300:

arquitetura	stack	accumulator	load-store
	lit 100		
parcela 1	load	load 100	load 100, src1
	lit 104		
parcela 2	load		load 104, src2
soma	add	add 104	add src1, src2, dst
	lit 300		
resultado	store	store 300	store dst, 300
instruções	7	3	4
ciclos	10	9	11 (8)
bytes	19	15	22

Register spilling

- quando os registos não chegam é necessário guardar valores temporários na memória, em geral, na pilha de dados:
- **Stack machine:** não tem reserva de registos nem *spilling*:
 - melhor densidade de código
 - compilador mais simples
 - difícil de paralelizar (*pipelining*)
- **Register machine:** compilador maior e mais complexo:
 - acesso mais rápido aos registos
 - uso eficiente dos registos pelo compilador
 - *pipelining* é determinante na eficiência

$$b^2 - 4ac$$

arquitetura	stack	accumulator	load-store
	lit 104	load 104	load 100, a
	load	mul 104	load 104, b
spill	dup	store t1	load 108, c
	mul	lit 4	load 4, d
	lit 4	mul 100	mul b, b, e
	lit 100	mul 108	mul d, a, f
spill	load	store t2	mul c, f, g
	mul	load t1	sub e, g, h
	lit 108	sub t2	store h, 300
	load	store 300	
	mul		
	sub		
	lit 300		
	store		
instruções	14	10	9
ciclos	18	30	22 (16)
bytes	34	50	46

Modos de endereçamento

- **imediato:** o operando é a instrução (lit 100)
- **direto:** endereço absoluto na memória (load 100)
- **aumentado:** concatenação de um registo base c/ bits baixos
- **deslocamento:** soma com registo base
- **indireto:** o endereço contém outro endereço
- **relativo:** relativo à instrução corrente
- **implícito:** implícito no nome da instrução

Custo das instruções

- **simples:** soma, subtração, deslocamento (shift).
- **complexas:** multiplicação, divisão, salto, chamada, retorno.
- **elaboradas:** chamadas ao sistema, interrupções.

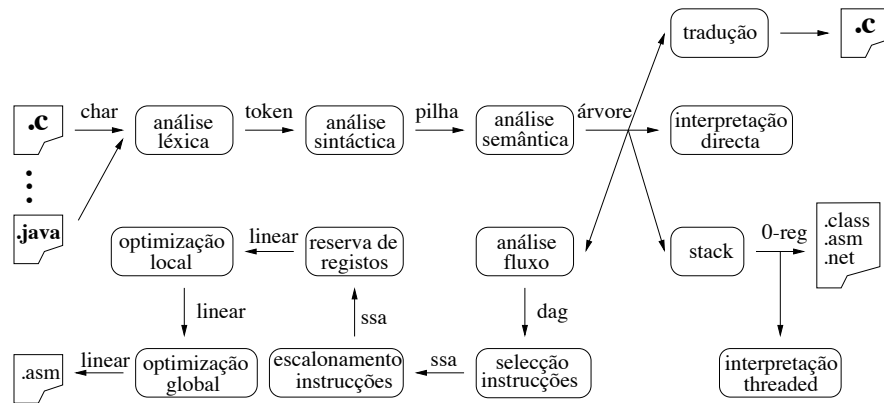
Considerar o tipo de endereçamento e o acesso aos parâmetros e resultado:

- **registo:** baixo custo. (1x a 2x)
- **cache:** uso frequente. (2x a 5x)
- **memória:** uso pouco frequente. (2x a 100x)
- **disco:** em ficheiro ou por paginação. (1000x a 10000x)

Geração de código

- seleção de instruções: uniformidade, velocidade, dimensão, modos de endereçamento.
- reserva e gestão de registos.

Compilador otimizador



The End

That's all folks!